10-1-2008

A novel partial reconfiguration methodology for FPGAs of multichip systems

Juan Manuel Galindo

A Novel Partial Reconfiguration Methodology for FPGAs of Multichip Systems

by

Juan Manuel Galindo

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science
in
Computer Engineering

Approved By:

Dr. Eric Peskin
Assistant Professor, Department of Electrical Engineering
Thesis Advisor

Dr. Muhammad Shaaban
Associate Professor, Department of Computer Engineering

Dr. Dhireesha Kudithipudi
Assistant Professor, Department of Computer Engineering

Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
October 2008

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: A Novel Partial Reconfiguration Methodology for FPGAs of Multichip Systems

I, Juan Manuel Galindo, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Juan Manuel Galindo

Date

Dedication

To my parents for their invaluable support and to my girlfriend for her patience and understanding during the course of this research.

Acknowledgments

I am grateful to have had corporate liaisons that supported my research and an advisor that took a special of interest in my thesis topic. I would also like to thank Professor Michael Eastman from the Computer Engineering Technology department for his practical insights that eased the feasibility of this research.

Abstract

A number of SRAM-based *field programmable gate arrays* (FPGAs) allow for *partial reconfiguration* (PR). Partial reconfiguration can be used to maximize the resource utilization in these FPGAs. Any large design usually consists of many modular features that are never used all concurrently. An FPGA does not need to implement all these features at the same time provided that it can be reconfigured in a reasonable amount of time to implement the features that can be used simultaneously. The use of partial reconfiguration is ideal in this case, since it allows for just the features that are no longer needed to be replaced by the newly required features. Current methodologies use both *external* and *self* partial reconfiguration for this purpose. On mature *multichip* (MC) systems that have not made use of the PR features of their SRAM-based FPGA(s), however, these methodologies would require changes in the existing FPGA configuration protocol and/or associated hardware outside the array.

This thesis presents a novel methodology that makes PR features available to these systems for the purpose of maximizing their FPGA resources without the modifications required by the current methodologies. The proposed methodology reuses an existing data interface to send the PR data to the array and directs this data to the FPGA's internal configuration port. A prototype of this methodology is demonstrated on a commercial *color space conversion* (CSC) engine design using two Xilinx Virtex-II Pro FPGAs. In addition, the effectiveness of the proposed methodology is quantified by comparing the FPGA resource utilization of the original CSC engine design and that of the partial reconfigurable prototype above. Finally, since the application of partial reconfiguration inherently adds latency to the output of any design, the effects of the proposed methodology on the performance of the CSC engine are also studied and reported. This information will show that reconfiguring and loading the prototyped CSC engine in addition to processing a full image in it takes 683ms, which is within the target of one second.

Contents

List of Figures

List of Tables

Glossary

C

CLUT Color Look-up Table - A device which converts the logical color numbers stored in each pixel of video memory into physical colors, normally represented as RGB triplets, p. 10.

CSC Color Space Conversion - The translation of the representation of a color from one basis to another, p. v.

H

HDL Hardware Description Language - Any language from a class of computer languages and/or programming languages for formal description of electronic circuits, p. 28.

I

ICAP Internal Configuration Access Port - The Virtex FPGA primitive that allows FPGA user logic to access the configuration memory of this FPGA device, p. 20.

ICI Internal Configuration Interface - A custom IP that monitors the partial reconfiguration process and drives the ICAP in the proposed methodology, p. 19.

L

LA Logic Analyzer - A device used to analyze concurrently a number of digital signals on a electronic system, p. 26.

M

MC Multi-chip - A electronic system comprised of more than one integrated circuit, p. 7.

P

PRM Partial Reconfiguration Module - A logic module that can replace a similar module in the FPGA fabric at runtime, p. 32.

PRR Partial Reconfiguration Region - An area on the FPGA floorplanned to host two or more PRMs, p. 24.

T

TB Test Bench - A virtual or hardware environment used to verify the correctness or soundness of a design or model, p. 25.

Chapter 1

Introduction

SRAM-based FPGAs (simplified to "FPGAs" in this document) are hardware devices that need to be reconfigured every time they are powered up. This type of reconfiguration is called static reconfiguration. Many FPGA-based systems boot their FPGA(s) using static reconfiguration but never change the configuration of their FPGA(s) at run time even though this *run-time reconfiguration* (RTR) support is available in some FPGAs and is referred to as *dynamic reconfiguration*. This latter reconfiguration type is further divided into two categories: *full* and *partial*. In full dynamic reconfiguration, all the hardware features in the FPGA are wiped out when the device is reconfigured. On the other hand, *partial reconfiguration* (PR) [26] is a unique feature of some FPGAs that allows the reconfiguration of a part of the array while the rest of the FPGA continues operating. Note that partial reconfiguration must be dynamic by definition. Furthermore, based upon the granularity of RTR required, two classes of partial reconfiguration approaches can be employed: (1) *difference based*, and (2) *module based* [26]. Difference-based reconfiguration methods are useful when attempting to perform a fine-grain dynamic reconfiguration while module-based ones are generally more suitable for coarse-grain dynamic reconfigurations.

This thesis presents a partial reconfiguration methodology that increases the functionality of FPGAs with scarce resources without bill of material changes. The proposed methodology only requires these FPGAs to be part of a mature multichip design. The main motivation for this research has been to devise a methodology that would allow for the functionality of the above FPGAs to be increased after deployment. Note that there are

PR methods and architectures that allow for a handful of MC systems with FPGA lacking resources to be upgraded once they are deployed. Hence, the aim of the proposed methodology is to be useful for most MC systems that have already been deployed but that require FPGA firmware upgrades. Other motivations of this research include the ability to improve time-to-market of new releases containing the increased FPGA functionality and to reduce the overall cost of new FPGA firmware releases. The proposed methodology achieves these results by allowing the incorporation of more logic to the existing FPGA firmware without the need to replace any hardware even if the FPGA resources are scarce.

In Chapter 2, the state of the art in PR technology is presented. Chapter 3 describes the proposed PR methodology in detail. It makes use of the Xilinx PR flow [11] and of the well-defined communication channels for both data and control between the FPGA and the integrated circuits outside the array. The components of the methodology are adaptable to user applications provided that they fall within the FPGA application domain described above (*e.g.*, large FPGA pipeline designs within MC systems). A demonstration of the module-based methodology is presented in Chapter 3, where a prototype is designed using the methodology discussed. This chapter also presents background information pertaining to the prototype application. Chapter 4 delves into the implementation of this prototype using two Xilinx Virtex-II Pro FPGAs. Chapter 5 discusses the tests performed on the prototype application. Chapter 7 summarizes the results of the prototype, including an analysis of reconfiguration times and resource utilization. Finally, Chapter 8 presents conclusions and future work.

Chapter 2

Background and Previous Efforts

Module-based partial reconfiguration has been an area of much interest in the research community [5, 7, 28, 30], since it only requires that a system be composed of modules that are never used all concurrently, which is usually the case in large systems. This type of reconfiguration allows modules to be loaded only when an application is invoked and removed again once the application has terminated [25]. The modules that are always required at different times are called mutually exclusive. The main benefit from this type of reconfiguration is that it minimizes the number of unused modules resident in the FPGA, which waste power, area, and cost.

2.1 Partial Reconfigurability: Benefits and Drawbacks

In practice, the viability of partial reconfigurability for the purpose of maximizing FPGA resources is subject to some additional requirements. Hardware-wise, the target FPGA design must have access to an external storage facility that stores the partial reconfiguration data to be sent to the FPGA at run-time. In addition, only SRAM-based FPGAs support partial reconfiguration. The key element that makes these FPGAs partially reconfigurable is their configuration architecture, which consists of three levels. On the highest level or routing level, the interconnects associated with the global resources of the FPGA, namely clocks, are programmed such that they are properly distributed in the FPGA. On the user logic or first floor level, FPGA primitives are connected to form the hardware components

of the FPGA design. The configuration or basement level contains the information regarding the connections on the user logic level. Changes in the configuration level directly affect the functionality of the user logic but not how the global resources are distributed. It is, in fact, the memory-based architecture of the configuration level of SRAM-based FPGAs that enables partial reconfiguration in these devices, since it allows a partial area of the user logic to be reconfigured by changing a section of the configuration memory. This research uses a Virtex II Pro SRAM-based FPGA to prove the feasibility of partial reconfigurability in a commercial application because the Virtex family in particular is capable of self-reconfiguring itself. It is capable of doing so because it has a primitive called the *internal configuration access port* (ICAP), which allows user logic to access the configuration memory of these devices. Moreover, the Virtex II Pro FPGA has an embedded processor, namely a PowerPC hard IP, which will be used to generate test vectors for the partially reconfigurable design of this research.

Implementation-wise, partial reconfigurability adds additional steps to the standard FPGA design flow, namely the partial reconfiguration flow [11]. For instance, partial reconfiguration is an FPGA process that cannot be captured in a behavioral model. Thus, the behavior of an FPGA design cannot be simulated during the partial reconfiguration process, which prevents the design from being fully verified at this stage of the partial reconfiguration flow. Furthermore, this flow does not support all the *hardware-description language* (HDL) coding standards supported by FPGA designs that do not employ partial reconfiguration. Chapter 6 discusses all the implementation considerations of the partial reconfiguration flow.

Admittedly, partial reconfigurability adds complexity and requirements to the development of a product. However, the benefits often surpass the drawbacks for some applications. For example, commercial applications in general require inexpensive products. Partial reconfigurability keeps the cost of commercial FPGA-based designs down by enabling more hardware functionality to fit on an FPGA. There are certainly other measures besides partial reconfigurability that can be taken to accomplish this goal such as choosing

inexpensive analog and digital parts for the design for instance. In the case of FPGA-based products that have already been deployed and that require more hardware functionality, however, partial reconfigurability is indeed the only mechanism through which these requirements can be met. One of the contributions of this research is an optimal methodology for using partial reconfigurability in this scenario in particular.

2.2 Previous Efforts

There are two types of module based partial reconfiguration (simplified to "partial reconfiguration" in the remaining of this document): *self* and *external*. In self-partial reconfiguration, the configuration data may be generated by the FPGA itself [18] or provided by an external source. In the latter case, an interface to a *configuration data medium* [12] (external to the FPGA) that provides a channel to the PR data offline storage facility [2, 6, 12] is required. On the other hand, external partial reconfiguration requires custom logic outside the array to drive the partial reconfiguration interface [19, 21, 29].

Partial reconfigurability can provide a number of advantages to an FPGA design including adaptive hardware algorithms, more efficient use of FPGA resources, and single-event upset correction among others. This research focuses on how to best enable partial reconfiguration for the purpose of maximizing FPGA resources. Thus, the purpose of the previous efforts on partial reconfigurability, which are summarized below, is also to achieve the most efficient use of FPGA resources. Note that if there are some mutually exclusive functional blocks within an FPGA design, external or self-PR can be used to make room for the additional hardware features required.

The release of the Virtex family of Xilinx FPGAs originated much of the initial research on partial reconfiguration platforms. For instance, Blodget *et al.* have reported on a *self reconfigurable platform* (SRP), where specific circuits on the array are used to control the reconfiguration of other parts of the FPGA [2]. Also, in an application note from Xilinx some of the primitives associated with the multi-gigabyte transceivers (MGTs) of a Virtex

FPGA are reconfigured on the fly by the associated embedded system implemented on the same FPGA [8]. Note that the Virtex's internal configuration interface is the ICAP.

A more recent approach to self-partial reconfiguration is the method proposed in [25] termed *merge dynamic reconfiguration*, where modules can be assigned arbitrary rectangular regions in the FPGA and relocated at run-time. Merge dynamic reconfiguration enables self reconfiguration through the SRP, which is composed of an internal configuration interface, control logic, a small configuration cache, and an embedded processor [2]. This reconfiguration method exploits the glitchless property of partial reconfiguration through reserved routing and minimizes the sizes of the partial reconfiguration bitstreams that are stored in FPGA user space memory by reading configuration memory content into user space memory and applying Boolean operators on the configuration bits read back.

The state of the art in self-partial reconfiguration for the purpose of maximizing FPGA resources is also an extension of the SRP platform. Hübner *et al.* describe a self-PR platform for Network-On-Chip applications [14]. In this platform, the FPGA requires access to the flash memory where the configuration data is stored. Bomel *et al.* have proposed a self-PR architecture in which the FPGA obtains the configuration data via an Ethernet connection to an external server [4]. Note that this connection could potentially be used to send processing data to the FPGA. Admittedly, existing self-PR methods [2, 3, 6, 12] including those described above, assume that the FPGA can obtain the configuration data via one of the following interfaces: RS232, IrDA, 10/100 Ethernet, or flash memory. An FPGA may not always be interfaced to these specific standards, however, especially in applications where the array is used primarily to implement custom logic. Self-PR methods that generate the configuration data inside the FPGA are not considered here because they are mainly used for fault tolerant applications.

On the other hand, the state of the art in external partial reconfiguration has not seen as much improvement over the last few years. The main reason for the lack of interest from the research community on external partial reconfiguration is its external logic requirements, since they do not allow for all the custom DSP hardware and control logic of an application

to be implemented on a single FPGA. These external logic requirements also make it more challenging to change the FPGA configuration scheme, which is important when a designer is trying to leverage partial reconfiguration on an existing FPGA design. The last published research on external partial reconfiguration is the work of Delahaye *et al.* [10]. Their platform uses a DSP and a dedicated interface to load the initial configuration in the FPGA and to partially reconfigure it as needed.

Traditionally, PR platforms and frameworks have been first devised and then applied to designs matching the requirements of these architectures. The lack of commercial products that use partial reconfiguration, however, shows that this approach ought to change. Active research in the area of partial reconfiguration is addressing this issue by delivering PR methods tailored to specific applications rather than generic PR architectures [22, 24]. The methodology proposed in this thesis is optimal for FPGA designs that are part of a MC system. Such applications constitute our target application domain. Chapter 3, and in particular Section 3.1, delves into the proposed methodology.

Chapter 3

Methodology and Application

The proposed PR methodology and a prototype system that uses it to maximize its FPGA resource utilization are presented in this chapter. Section 3.1 presents the requirements and components of the proposed methodology. Section 3.2 provides some background material on a conversion engine design that is used by a commercial application. Section 3.3 presents the components of the prototype system, which include this design.

3.1 Methodology

This section presents the need for the proposed PR methodology as well as its requirements and components. The proposed methodology is optimal for MC designs that require additional features on their FPGA but there are not enough resources and the *bill of materials* (BOM) cannot change. In general, board layout changes are highly undesirable in mature MC designs, since they may affect the operation of working components. Moreover, the addition of FPGA functionality cannot require BOM changes in systems that need to be updated from the field.

A key benefit of the proposed methodology is that it can be applied without BOM changes even if the FPGA application was not originally designed for PR. If external FPGA configuration hardware that does not support PR already exists for some FPGA design, any external PR architecture [19, 21, 29] including Delahaye's platform in this design would require BOM changes. In addition, the proposed methodology reuses any data interconnect

for the FPGA to receive the configuration data. Hence, BOM changes can be avoided more often in the proposed methodology than in the existing self-PR methodologies including Hübner's and Bomel's which require that the FPGA be connected to specific interfaces.

The proposed methodology requires a partially reconfigurable design and internal access to the FPGA configuration port. In these designs, the top-level module implements only the global logic and the connections amongst other PR components. These components include bus macros, static modules, and *partially reconfigurable modules* (PRMs). On the other hand, internal access to the configuration port is required so that the logic needed to drive this port can be implemented on the FPGA configurable logic.

The proposed PR methodology consists of a technique called *interconnect timesharing* and a custom *intellectual property* (IP) called the *internal configuration interface* (ICI). Interconnect timesharing allows reconfiguration data to be sent via an interconnect that is already being used by the user logic hardware. The ICI block simply directs the already expected reconfiguration data to the internal configuration port of the FPGA. Figure 3.1 provides a graphical representation of the proposed methodology, where static modules are placed in the static region and PRMs in the *partially reconfigurable region* (PRR) as outlined in [26].

Interconnect timesharing requires that the functionality of the existing control logic be augmented to work as follows. First, as part of the configuration of the user logic the external device informs the augmented control logic that reconfiguration data will be sent before payload data. The control logic forwards this information to the user logic and the ICI block. The reconfiguration data count is also sent to the control logic in this initial step, which forwards it to the ICI block to determine when to signal the end of the reconfiguration process. Then, when data is sent to the FPGA via the shared interconnect, the user logic ignores this data since reconfiguration data is expected. On the other hand, the ICI block samples this data until the end of the reconfiguration process is reached. The control logic then signals the user logic that it may begin sampling from the shared interconnect. It is important to mention here that the proposed methodology does handle the case where PR

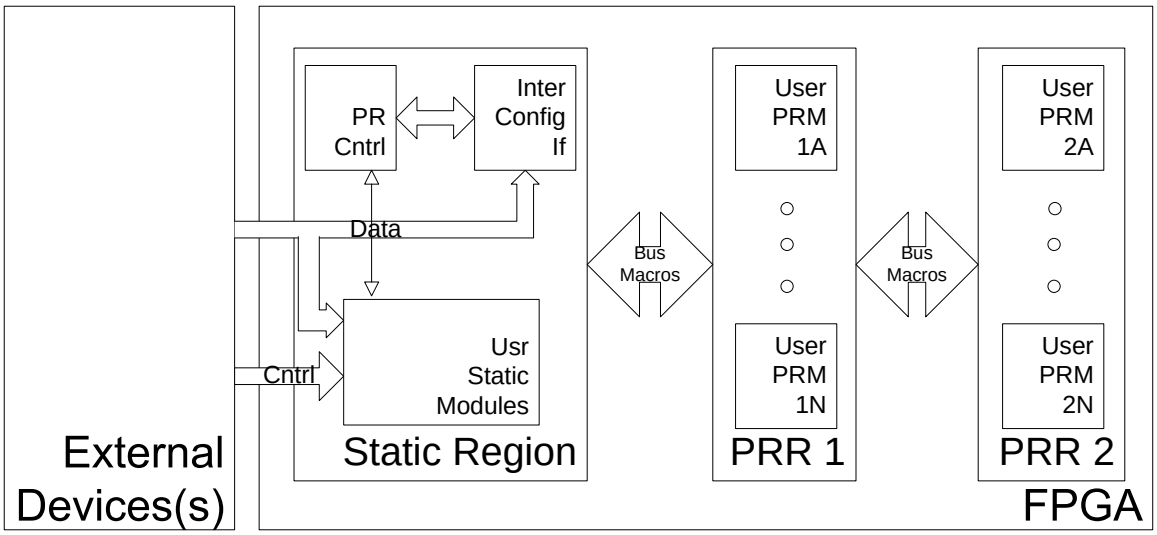

Figure 3.1: Proposed methodology framework.

is not required after the user logic is configured via the control logic.

The ICI block is composed of two small *finite state machines* (FSMs) and additional glue logic. This interface drives the control signals of the FPGA's internal configuration port as reconfiguration data is received via the shared interface.

3.2 Application Background

Color space conversion (CSC) is the translation of the representation of a color from one basis to another [13]. Hewlett Packard (HP) employs a hardware-based CSC engine to convert pixel color data formats into values corresponding to the colors of the inks loaded in their printers. The HP CSC engine uses *color look-up tables* (CLUTs) followed by interpolation [17]. It is composed of a fifteen-stage pipeline and SRAM memory that function primarily as interpolation blocks and CLUT tables respectively. In order to define these and other functional blocks within this engine, pipeline stages and SRAM memory are grouped into phases. Each phase contains up to three pipeline stages. These phases, however, are never all active simultaneously. In particular, depending on the type of conversion, one of the two largest phases is always bypassed. These phases are called the 3D and the 4D. The

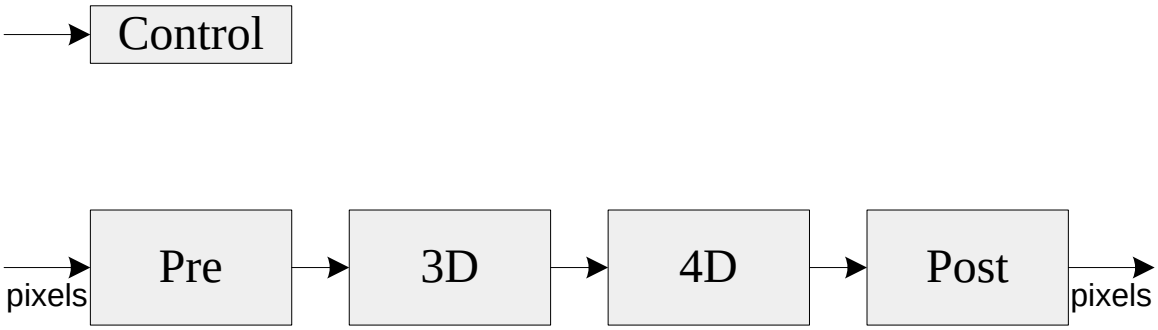

Figure 3.2: Core of the CSC engine.

3D phase is used when the input space has three channels. The 4D phase is used when the input space has four channels. Each of these two phases consists of three pipeline stages and over 40KB of SRAM. Figure 3.2 illustrates the core of the CSC engine.

In addition, the top-level IO of the CSC engine includes a pixel and a register interface through which pixel color data formats and CLUT values are sent to this engine. Color space conversion along with pixel scaling and halftoning are the main elements of any printer's image processing pipeline. Currently, HP implements this pipeline on a single *application specific integrated circuit* (ASIC). However, an HP printer is comprised of at least this ASIC, a memory subsystem, and an embedded processor. The latter components are employed to send data to the CSC engine and to implement the printer's user interface logic. This fact makes HP printers MC systems.

3.3 Prototype System

The CSC engine is implemented as part of an ASIC in commercial HP printers. Using this as the driving example for our research requires two changes. First, we resynthesize the engine for an FPGA. Then we introduce PR into the FPGA implementation using our methodology. Due to this difference in target devices, a new PCB board that accommodates the FPGA package implementing the ASIC functionality would be required to use the actual components that drive the CSC engine. Instead, these components are replaced by very similar off-the-shelf ones. This is a key aspect of the prototype system. Admittedly, the

successful modification of the firmware run by the actual components of a printer to send the PR data to the FPGA-based CSC engine prototype would have truly demonstrated the interface timesharing technique. Nonetheless, the successful modification of the firmware run by similar hardware is also a valid demonstration. Note that if the commercial design had already employed an FPGA, our methodology would allow adding PR to the design without changing the board or interface. This is an important feature of our approach.

Two of the largest modules of the CSC engine, namely the 3D and the 4D phase, are mutually exclusive. Because each of these phases require a third of the slices available in the FPGA used in the prototype system, our methodology maximizes the use of these slices by swapping the 3D and the 4D modules as they are needed to convert pixel data.

The interconnects attached to the CSC engine's register and pixel interface are used by the modified firmware above to initiate the PR process and to transfer the PR configuration data respectively. The augmented control logic designed for this prototype consists of a new register that manages the PR process inside the FPGA and that siphons PR data off of the pixel interface as required. The internal configuration interface IP, which is connected to this register and to the pixel interface, drives the ICAP of the Xilinx FPGA used in the prototype. Finally, additional logic is employed to stall the CSC engine pipeline while partial reconfiguration is being performed by the ICI block.

Chapter 4

CSC Engine Implementation

An FPGA-based implementation of the CSC engine, where the proposed methodology is used to maximize the array resources, is presented in this chapter. Section 4.1 discusses the hardware blocks required to prototype and test an FPGA-based CSC engine. A production candidate of this prototype would not contain the modules discussed in this section. Section 4.2 delves into the modifications required by an FPGA-based implementation of the CSC engine that intends to use partial reconfiguration to maximize the array resources. In addition, Section 4.3 presents the implementation flow used in the prototype system. A production candidate would use this same flow. Finally, Section 4.4 describes the firmware modifications performed in the prototype system in order to send the PR data to the FPGA implementing the CSC engine without having to change the FPGA configuration protocol and/or hardware outside the array. Moreover, this section argues that these modifications are also possible in the components that drive the CSC engine in commercial HP printers.

4.1 Prototype-Specific Hardware

Two development boards from the Xilinx University Program (*i.e.*, two XUP2VP boards), have been employed to implement the prototype system described in Chapter 3. One of these boards, called *the CSC board* from now on, implements a modified version of the original HP CSC engine. The other board, called *the TestRig board*, emulates the components that drive the CSC engine in an HP printer. A large number of FPGA *general-purpose*

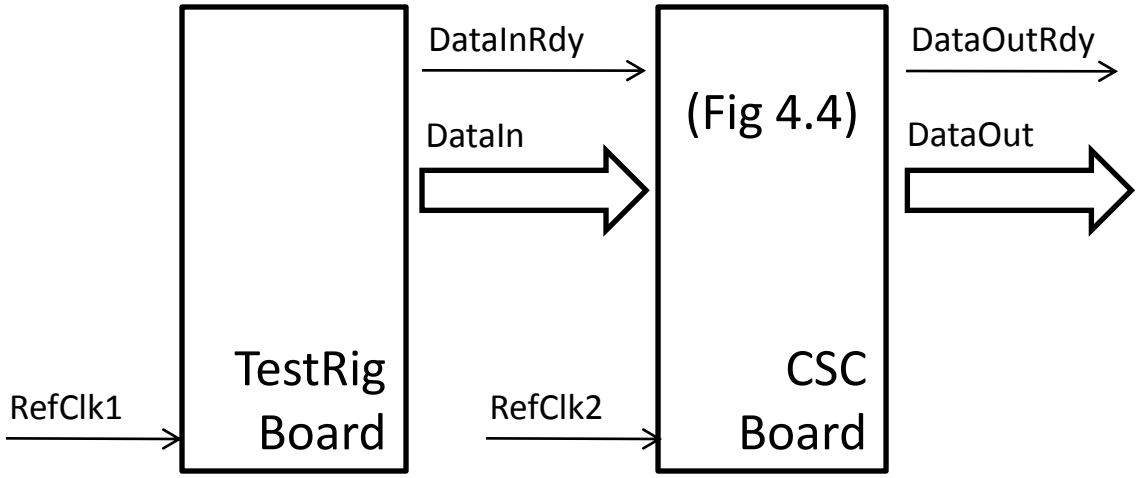

Figure 4.1: Prototype hardware.

input and output (GPIO) lines are available in these boards through standard IDE headers. These lines are used here to transfer data from the TestRig board to the CSC board. Note that the Virtex II-Pro FPGAs on these boards and both a *compact flash* (CF) card and *double data rate* (DDR) memory on the TestRig board are the only XUP board components used in the prototype system.

There are two reference clocks in our prototype system, which is illustrated in Figure 4.1. The DataInRdy signal is used to synchronize the TestRig and the CSC board. This TestRig board output signal is synchronized to the reference clock of the CSC board using three flip-flops in series as shown in Figure 4.2. This synchronization process also synchronizes the DataIn bus in the CSC board, since in the TestRig board the DataInRdy signal is asserted only when the DataIn bus is already valid. The DataOutRdy signal is used as a reference clock by the logic analyzer in charge of sampling the output of the CSC engine.

In the TestRig board, the PowerPC processor embedded in the Virtex II Pro FPGA generates the test vectors required to stimulate all the inputs to the CSC engine. This 32-bit processor limits the width of the digital connection between the TestRig and the CSC board to 32 bits. Since the CSC engine has more than 32 inputs, *CSC packets* that stimulate all the CSC inputs in a cycle-by-cycle basis are created in the TestRig board. Figure 4.3

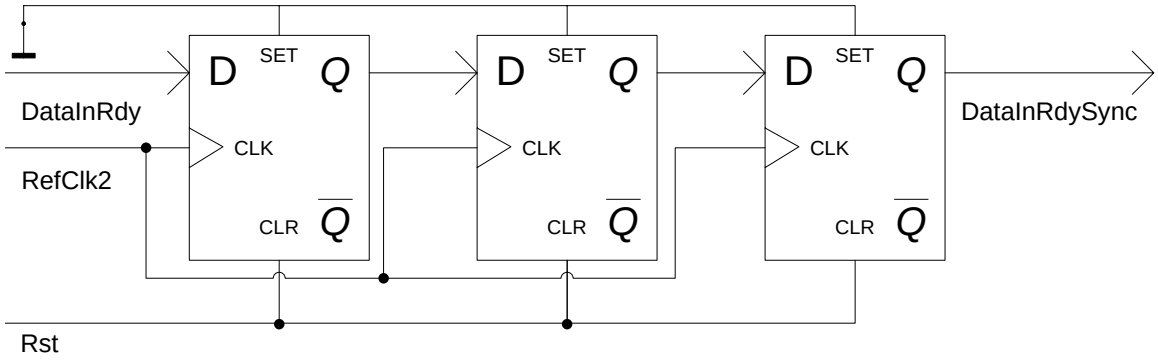

Figure 4.2: CSC board synchronization logic.

depicts the format of these packets. Note that four 32-bit transfers are required to populate all the inputs to the CSC engine. The first two transfers populate the control signals and the register interface of the CSC engine. The last two transfers populate the pixel interface of this engine. The register interface is affected by the CSC packets even when the engine is converting pixel data. Conversely, the pixel interface is populated by the CSC packets even when the CLUT values are being loaded through the register interface. This approach simplifies the interface between the TestRig and the CSC board.

In the CSC board, the data in these CSC packets needs to be depacketized and sent to the CSC engine control signals, register and pixel interface. Therefore, an additional depacketizer module that wraps the CSC engine interface and that depacketizes the incoming data from the TestRig board is also required in the FPGA prototyping the CSC engine. Figure 4.4 illustrates the depacketizer module. Note that the CSC black box in this figure has the same interface as the original HP design. The control logic associated with this module is shown in Figure 4.5. This module essentially shifts the incoming data from the TestRig board into a bit-parallel, word-serial shift register. Each word is 32 bits wide, and there are four words. Every four shifts, this module also signals the CSC engine that there is a new set of input vectors ready to be processed. A delayed version of the CSCClk signal, which is asserted every four shifts by the depacketizer module, is used as a gated clock in the CSC engine. Additional details in regards to the implementation of the TestRig and the CSC board can be found in Appendix A and Appendix B respectively. Note that

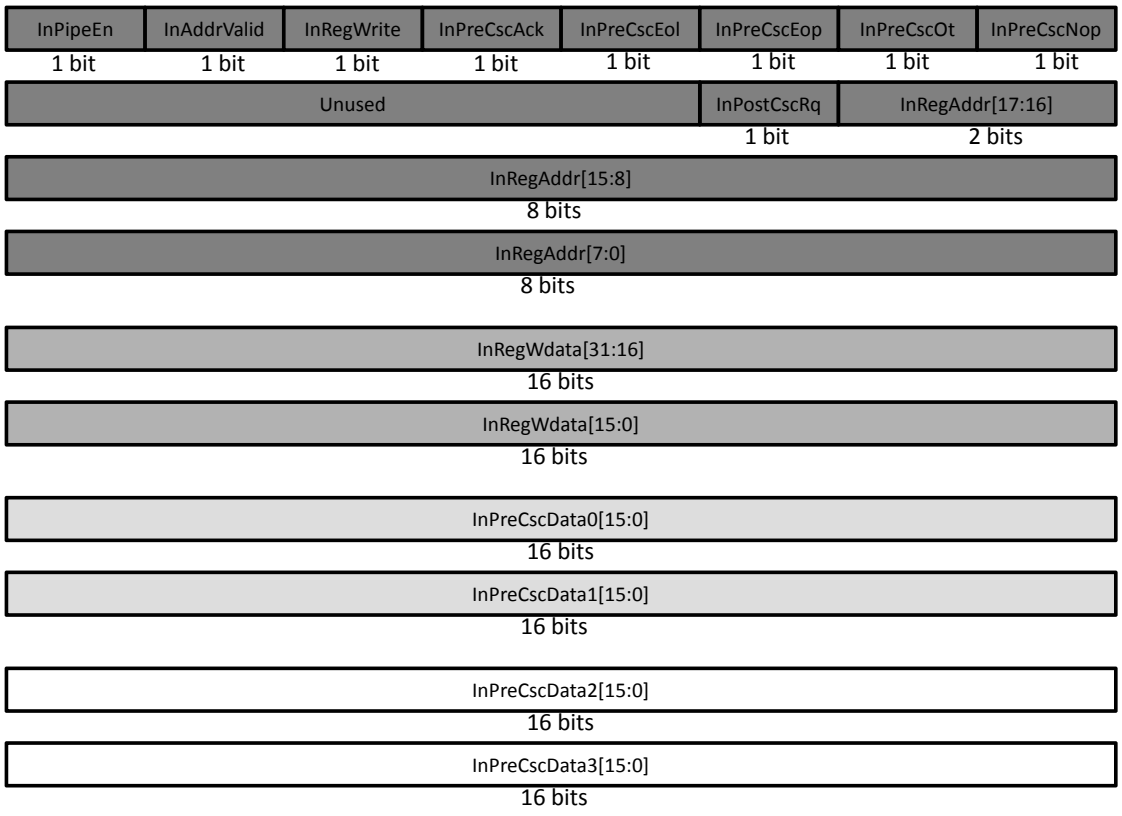

Figure 4.3: CSC packet format.

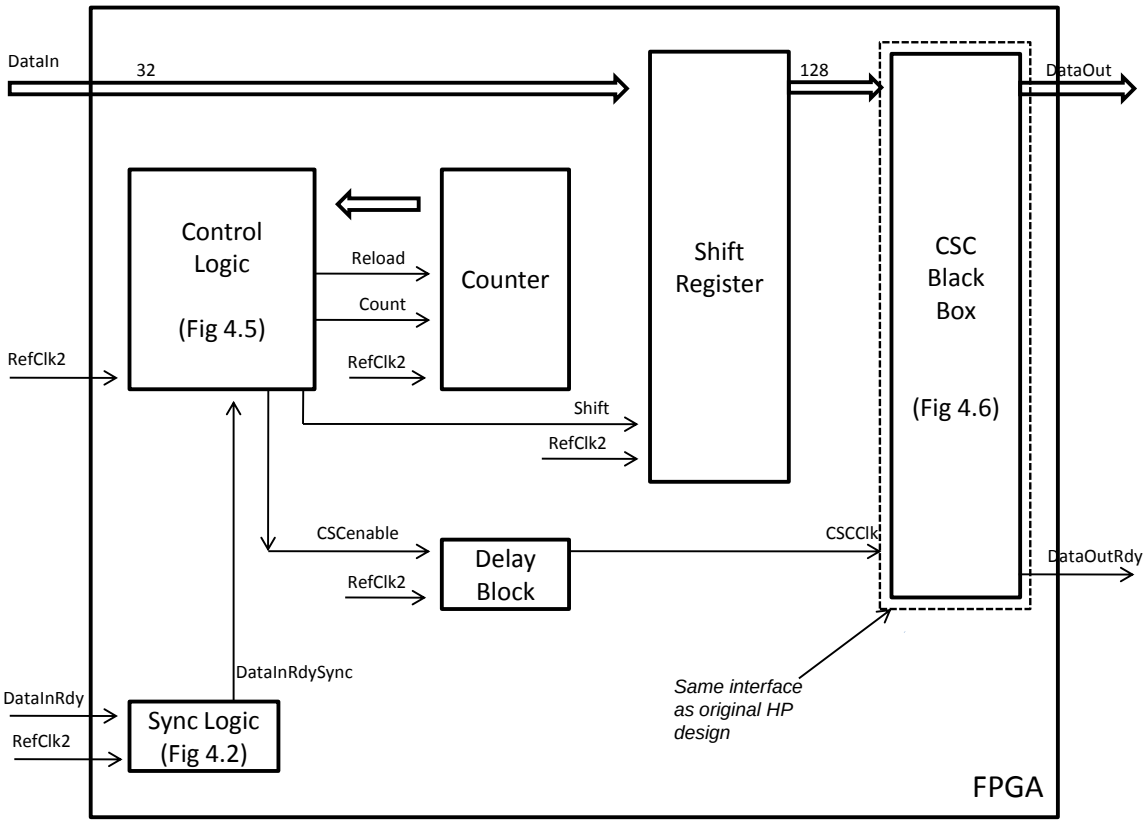

Figure 4.4: CSC board depacketizer block diagram.

a commercial implementation of the FPGA-based CSC engine would not require the depacketizer module above, since the inputs to the CSC engine would be driven by the other hardware components of a commercial printer. These components would fully drive the wide interface of the CSC engine.

4.2 Target-Independent Hardware

The original HP CSC engine which includes both the 3D and the 4D phase in addition to the other phases of the CSC pipeline does not fit on the FPGA of the CSC board. The prototype system leverages the fact that the 3D and the 4D phases are never active at the same time in order to fit the functionality of the CSC engine in this FPGA. It implements a modified version of this engine which uses partial reconfiguration in order to swap in and

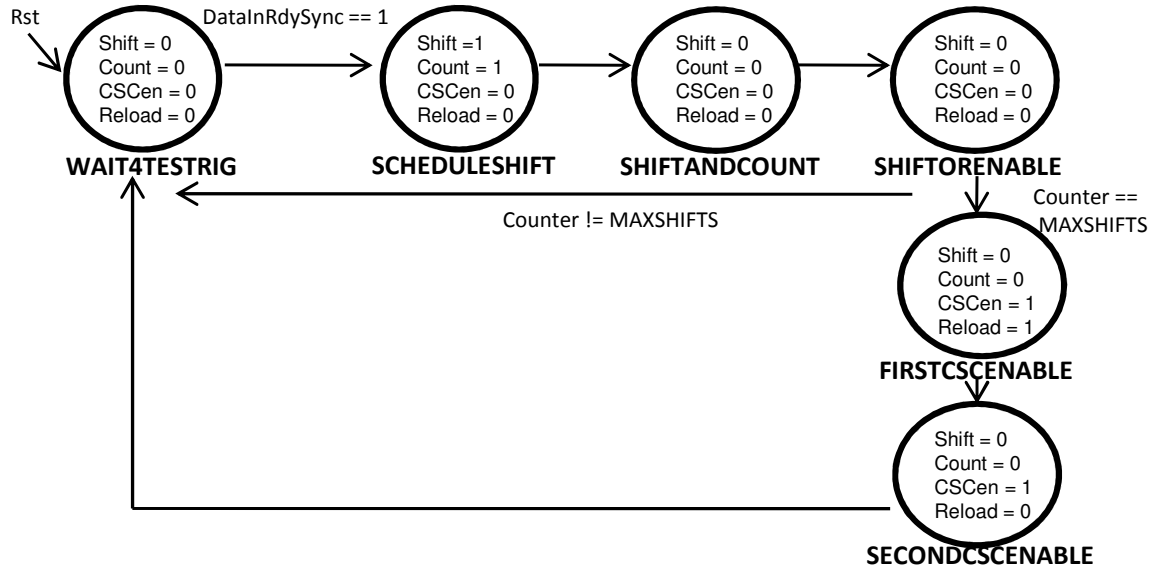

Figure 4.5: CSC board depacketizer state machine.

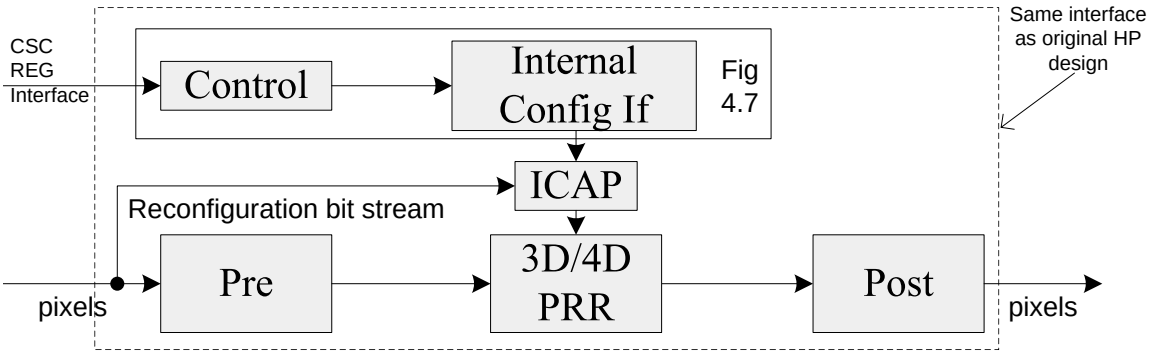

Figure 4.6: Modified CSC engine using PR.

out the 3D or the 4D phase from the FPGA as required. Figure 4.6 illustrates the design. Note that the modifications introduced in the prototype system do not change the interface of the original HP design.

A block diagram of the modules that have been either modified or added to the CSC engine in order to support partial reconfiguration is shown in Figure 4.7. The logic associated with the PR controller is implemented in the CSC Reg and Autoload modules. The *internal configuration interface* (ICI) is implemented as an independent functional block. In addition, minor modifications to the CSC Control and Autoload modules are required to

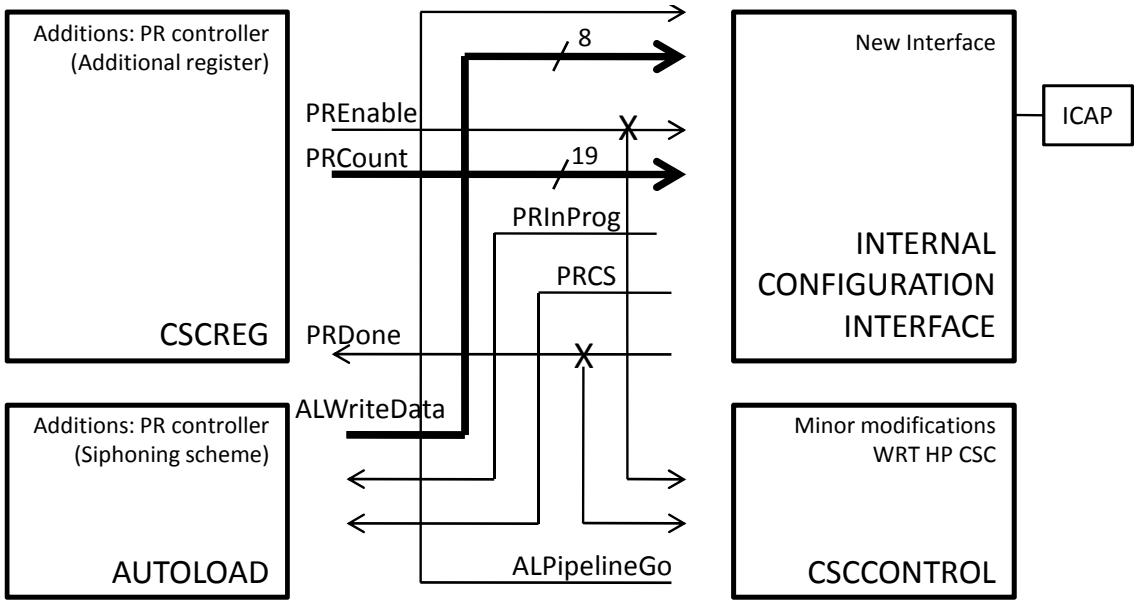

Figure 4.7: Modified and new CSC engine modules to support PR.

stall the CSC engine pipeline while partial reconfiguration is taking place.

Because of its interface to the control interconnect outside the array, namely the register interface, the CSC Reg module implements part of the PR controller. In an effort to maximize resource utilization, the rest of the PR controller functionality is implemented in the Autoload module, since this latter module has the remaining features required by the PR controller. These features include logic both to stall the CSC pipeline and to siphon data off of the pixel interface. The Autoload module requires these features to use the pixel interface to load the CLUT values.

Implementing the PR controller as part of the original HP design modules minimizes the changes required in the CSC Control module, since this module determines when the functional blocks within the engine should be enabled in a cycle-by-cycle basis, and admittedly the number of functional blocks has not been changed by the addition of the PR controller. The addition of the ICI requires only minor modifications in the CSC control module, since the ICI is active whenever the Autoload is active. The only design consideration required by this change is that additional logic in the Autoload module is needed to

ensure that the Autoload logic is disabled when the ICI logic is performing partial reconfiguration. Since the Autoload and the ICI features are activated at the same time by the CSC control module, this additional logic needs to be combinational.

The ICI module interfaces with the following CSC modules: Reg, Autoload, and Control. All these modules from the original HP design are used to control the pipeline. The part of PR controller that is implemented in the CSC Reg module signals the ICI when to expect PR data in the pixel interface. This logic also provides the ICI with the count of how many bytes to send to the ICAP. The other part of the PR controller, which is implemented in the Autoload module, passes on the siphoned PR data to the ICI. In addition, an interface to the Control module is required so that the ICI module is synchronized to the Autoload module, which is the module that provides the PR data to the ICI.

A timing diagram of the ICI top-level signals is shown in Figure 4.8. The ICI module has two input signals and two input buses. The PREnable signal and the PRCount bus are used by the CSC Reg to enable partial reconfiguration and to pass the PR byte count to the ICI. The ALPipelineGo signal is used by the CSC Control to enable the Autoload and the ICI hardware. The ALWriteData bus is used by the Autoload hardware to pass the siphoned PR data to the ICI. Furthermore, the ICI module drives three output signals. When the PRInProg signal is asserted, the Autoload hardware sends the siphoned data from the pixel interface to the ICI. While the PRCS signal is asserted, the Autoload hardware stalls the CSC engine pipeline. Finally, the PRDone signal is used by the CSC control to both disable Autoload and enable the pipeline. The state machine that drives these output signals is illustrated in Figure 4.9.

As mentioned earlier in this section, the ICI also controls the ICAP. Figure 4.10 illustrates a timing diagram of the signals associated with this control logic. Since the PRCS signal described above is asserted while there is valid data in the PRData bus, the state machine that controls the ICAP essentially asserts the proper ICAP control signals when the PRCS signal is asserted. This state machine is shown in Figure 4.11. Note that enabling the ICAP takes two clock cycles. Therefore, the data in the PRData bus needs to be delayed by

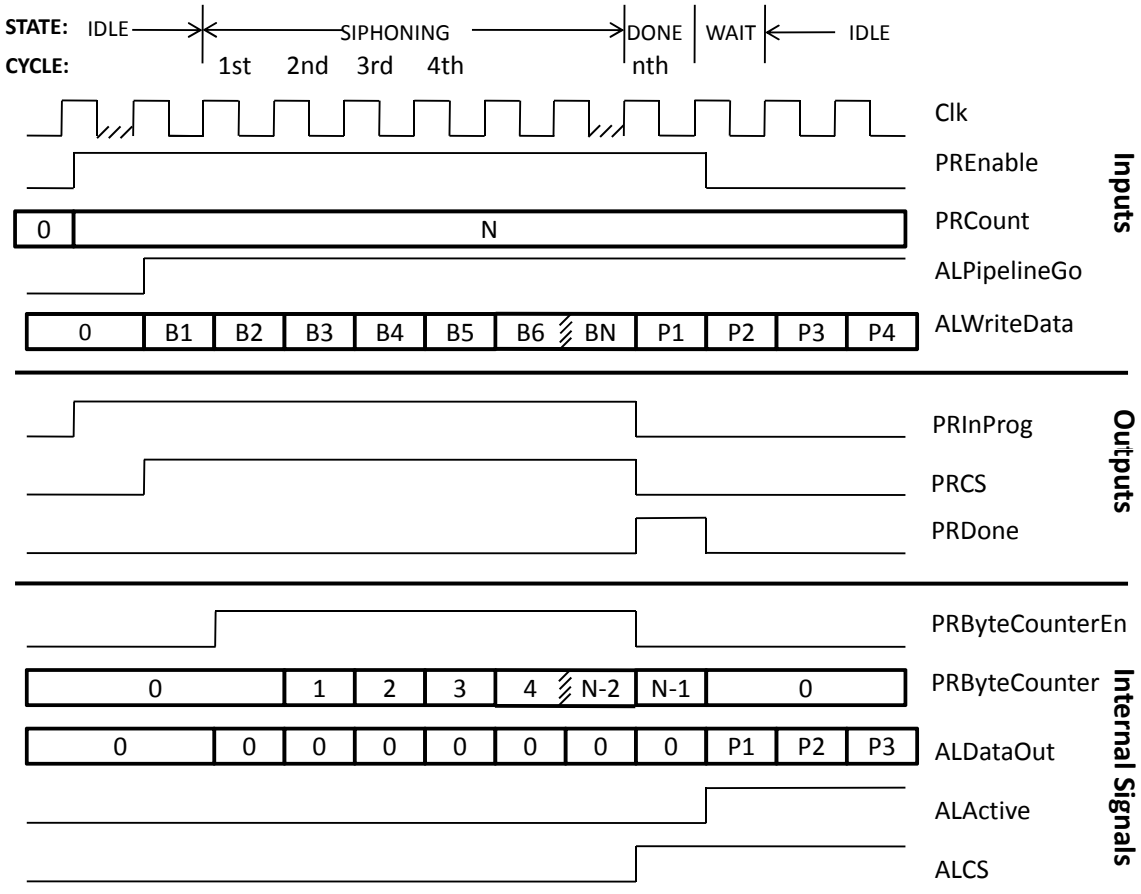

Figure 4.8: ICI timing diagram.

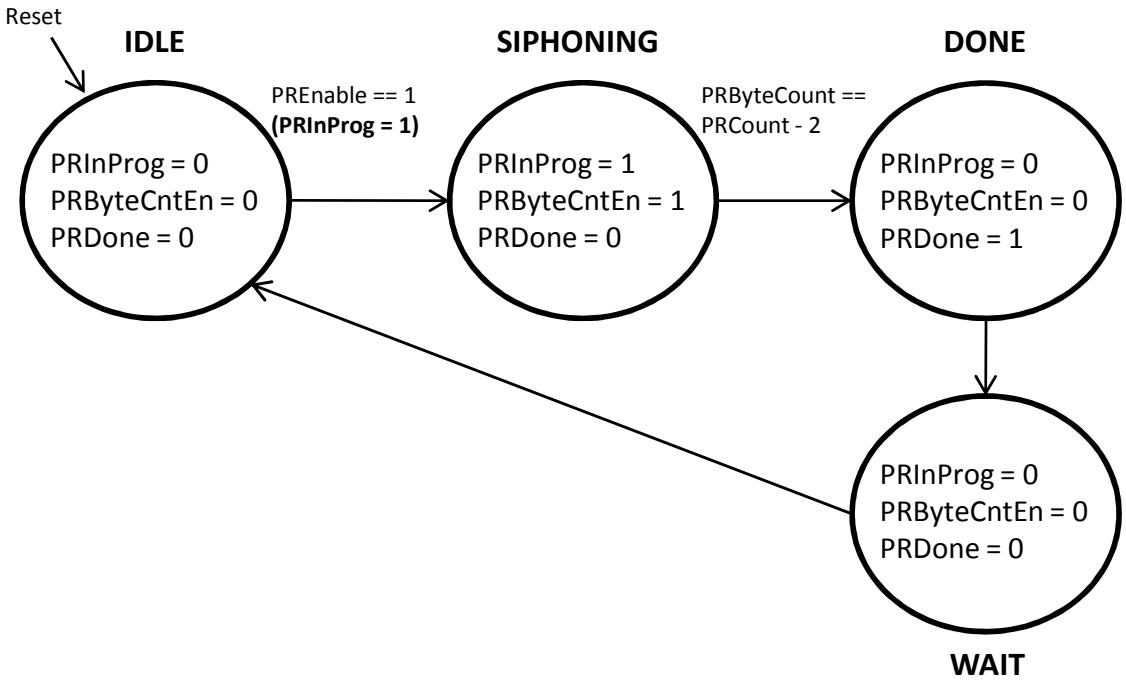

Figure 4.9: ICI state machine.

two clock cycles also before it is sent to the ICAP.

4.3 Target-Independent Implementation

A static or standard FPGA design requires its associated HDL sources files to be synthesized just once. On the other hand, partially reconfigurable designs require their source files to be synthesized at least twice in order to generate the different FPGA configurations that define partial reconfigurability. Note that an HDL project is required in order to synthesize a set of HDL source files. In this research, two module-based PR projects implement only one of the mutually exclusive CSC engine phases, namely the 3D or the 4D, as their PRMs and both the rest of the CSC pipeline phases and the logic required by the proposed methodology in their static regions. In the prototype system, the packetization logic is implemented in the static region also. Bus macros need to be used to connect the

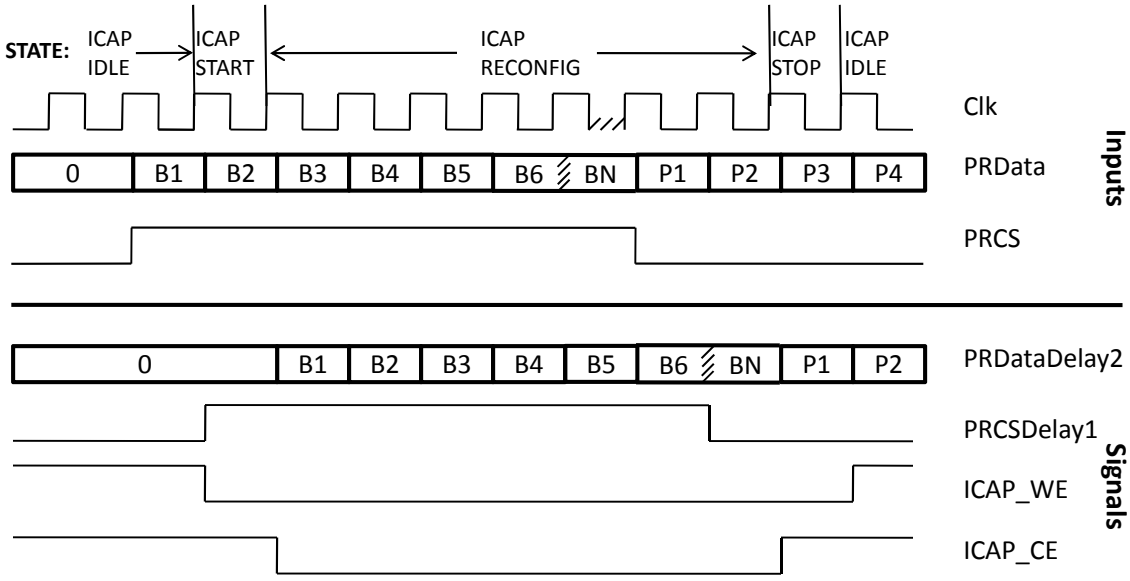

Figure 4.10: ICAP control logic timing diagram.

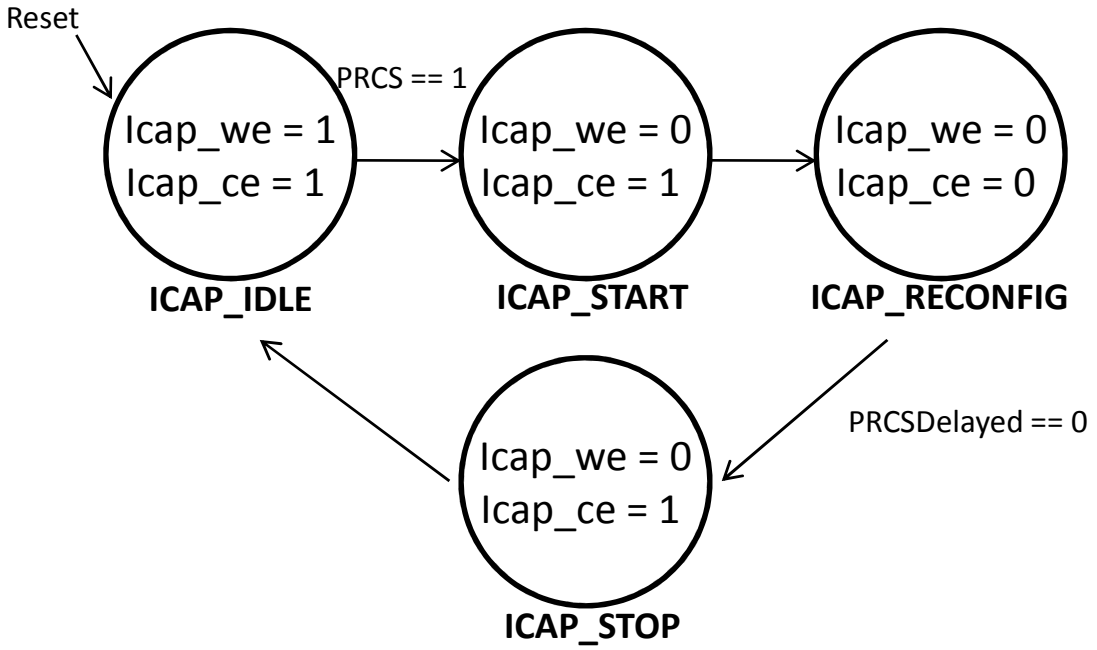

Figure 4.11: ICAP control logic state machine.

PRR to the rest of the design in the top level [11, 26]. For behavioral simulation, we replace each bus macro by a simple module consisting of pass-through wires. This allows the behavioral models of both PR projects to be verified using a simulator such as ModelSim. However, since behavioral models cannot change at run-time, the logic associated with the proposed PR methodology cannot be verified in behavioral simulation. This logic can only be verified through a hardware test bench like the TestRig board described in Sections 4.1 and 4.2.

4.4 Firmware Implementation

Finally, firmware for the TestRig board's embedded processor is developed in order for this board's hardware to be able to initially just send CLUT values and conversion data to the CSC board. Had the commercial embedded processor and DMA controller been used, this effort would not have been necessary. The final modification in this application of the proposed methodology is for the TestRig board's firmware to be enhanced so that its hardware is also capable of initiating the PR process and sending the PR data through the interconnects connected to the CSC engine's register and pixel interface respectively. These firmware enhancements are also possible in commercial printers, since here an embedded processor also drives the interconnects above either directly or indirectly. An example of the indirect control of the commercial processor over these interconnects is the fact that its firmware programs the commercial DMA controller to transfer pixel data to the CSC engine. In conclusion, the interface timesharing technique remains, where firmware modifications are the only requirement for sending PR related information to the CSC engine through both the TestRig and the commercial components that drive the CSC engine.

In this driving example, PR features are used to fit a large design in a particular target device. Note that partial reconfiguration, and in particular the proposed methodology, can also be used to incorporate new features into existing designs, provided that these features are mutually exclusive with at least a module of the original design.

Chapter 5

Prototype Testing

In addition to the CSC design, HP has also provided us with the CSC application. This application is a software program that emulates the hardware-based CSC engine. It can perform any conversion that is supported by the hardware-based CSC engine. One input to this CSC application is a file that contains the configuration required in the CSC engine phases for the desired conversion. A second input is a file that contains the CLUT values required for the desired conversion. The final input to this program is the image file to be converted. The output is the same image converted to the target different color space.

In commercial HP printers, these inputs are sent to the CSC engine as follows. First, the configuration of the CSC engine phases and the CLUT values are sent through the register interface if a new conversion type is to be performed. Then, the pixel data is sent through the pixel interface. Due to the packetization required in the prototype system, a *CSC frame* is defined here as a group of CSC packets that instruct the CSC engine to load its CLUTs and process an image. In order to build this frame in the TestRig board, a new *test bench* (TB) file that combines both of the CSC application input files as well as the input control signals of the CSC engine in a cycle-by-cycle basis is required.

As an initial test case, the firmware on the TestRig board first reads this new TB file from a CF card, parses it, and stores it in DDR memory as a CSC frame. Then, a user request on the TestRig board initiates the transfer of this frame to the CSC board. The output data bus of the CSC design is routed to the CSC board's FPGA GPIO lines. This allows a *logic analyzer* (LA) to collect the output data of the CSC engine while a CSC

frame is being sent from the TestRig board.

Since 33 out of the 80 GPIO lines available in the expansion headers of the CSC board are used to receive data from the TestRig board, only 47 lines are available to capture the output of the CSC engine in a LA. However, 50 signals need to be sent to a LA for this purpose including four 12-bit output pixel data channels and two control signals. One of these control signals provides the LA with an external reference clock so that it can be synchronized to the CSC board. The other signal called PostCscAck_Out is used by the LA to store only valid output pixel data. To overcome this minor issue, multiplexers are added to the output of the pixel interface in the CSC design such that either channel 0 and 1 or channel 2 and 3 are routed to the expansion headers of the CSC board depending on the logic level of an external switch. Therefore, the conversion of any image in the prototype needs to be executed twice in order for a LA to collect all the output associated with such a conversion. Note that the LA used here is a Tektronix TLA7N4 136 Channel, 64K, 100MHz, State Logic Analyzer Module.

Furthermore, the PostCscAck_Out signal is asserted by a conversion and stays active until a signal resets the CSC engine pipeline. This reset signal is routed to an external switch in the prototype. Since the LA will store data whenever the PostCscAck_Out signal is active, this external reset switch needs to be toggled every time an image is processed in the prototype so that only valid output pixel data is collected. Note that the aforementioned reset signal is a user logic signal. Therefore, this reset requirement does not affect the configuration of the FPGA in any way.

Finally, the output collected in the LA is merged and brought into a MATLAB environment along with the output produced by the CSC application of the same conversion. A MATLAB program simply performs a bit-wise comparison between these two output files to verify the correct functionality of the modified FPGA-based CSC design.

In an additional test case, the TB file is expanded to include PR data within it. To send this additional data to the CSC engine, the register interface is used to inform the CSC engine that the following data in the pixel interface is PR data. Then the PR data is sent

through the pixel interface. This reconfigures the PRR with either the 3D or the 4D phase as required. Finally, the CLUT values and the pixel data are sent through the register and pixel interface respectively. Note that the structure of the CSC packet does not change to accommodate the PR data. In other words, the existing CSC engine interface does not need to be modified in order for the PR data to be sent to this engine, which is indeed one of the main contributions of the proposed methodology.

For testing, we use two color-space conversions. One requires the 3D phase and the other requires the 4D phase. Back-to-back 3D and 4D conversions are performed in the prototype system using the above approach. Finally, the output of these conversions is compared against the output of the CSC application using the MATLAB program above to verify the functionality of the PR logic of the FPGA-based CSC design.

Chapter 6

PR Flow Considerations

The purpose of this Section is twofold. First, best practices for coding HDL designs that can be implemented either on an ASIC or a PR-enabled FPGA are described. Then, the practical issues associated with the implementation of PR designs using the Xilinx PR flow [11, 26] are reported.

6.1 Coding PR-Adaptable HDL Designs

There are three *register-transfer-level* (RTL) coding steps to take advantage of both ASIC and PR-enabled FPGA features. The first step is to minimize the speed degradation that a design encounters when its ASIC implementation is migrated to an FPGA target. An application note from Altera [1] shows how simple design considerations can help in creating designs that are optimal for both ASIC and FPGA implementations. For instance, logic duplication is a design attribute that can be assigned to functional blocks that have large fanouts in order to minimize routing delays in an FPGA implementation. The HDL source file can be written in a way such that this attribute is enabled when implementing the design in an FPGA and disabled when targeting an ASIC.

The second step aims to create designs that can be implemented in an ASIC but that are already PR-compliant. The PR flow [26] identifies many of the design considerations that are applicable in this step, since they do not affect the ASIC flow, including the pursuit of module-based designs and the use of global resources in the top-level of a given

design. An additional design consideration is to evaluate the use of generics in such a design. To ease their portability across different digital designs, commonly used modules are often implemented using *generics* [15, 16]. Generics are used to construct parameterized hardware components. Parameterized components make it possible to construct standardized libraries of shared models. However, since these models and the top-level module that instantiates them are synthesized independently in the PR flow, the use of generics cannot be supported in this implementation flow. A workaround for this portability issue exist in HDLs such as Verilog [16] or VHDL [15]. First, the parameters common to the top-level and the commonly used modules ought to be defined in a Verilog *header* file or a VHDL *package* file. Then, these files just need to be included in all the modules that are synthesized in the PR flow.

The final step is to create designs that facilitate the incorporation of the proposed methodology to enable PR. A register-based design would provide many of the features required by the PR controller. Also, an interface to non-volatile memory would provide the proposed methodology with a configuration data medium to store partial bitstreams. Finally, if a mechanism for freezing the output of the modules that are mutually exclusive with other modules within a design is supported, this feature can be used during the incorporation of the proposed methodology to prevent unknown states at the output of the PRR while partial reconfiguration is taking place.

6.2 Adapting HDL Designs to the PR Flow

There are several considerations that should be taken into account when porting a design to the PR flow including the following port mapping and the bus macro requirements.

A PR design has special rules for instantiating modules, bus macros and the ICAP. HDL port maps in the top level should produce *black boxes* during synthesis. These black boxes capture the IO signals of the modules instantiated in the top-level. This information is used to link the static and PRM modules to the top-level module during the implementation of

the PR design. Since HDL modules always need to be declared in a VHDL-based design, black boxes will be automatically generated during the synthesis of a top-level VHDL file. In a Verilog design, however, the generation of these black boxes needs to be enforced by declaring the IO of the modules instantiated in the top level within the top-level file itself. These module declarations need to be made with proper care, since any discrepancies between these declarations and the implementation of these modules will not be visible when the top-level Verilog module is synthesized. Potential discrepancies include signal direction and bus width mismatches.

Furthermore, pass through wires rather than bus macros should be initially instantiated in the top level so that the entire PR design can be simulated using a behavioral simulator such as ModelSim. Once the functionality of the design is verified, these wires can be replaced with the actual bus macros. It should be noted here that bus macros are architecture-specific. Moreover, there a couple of considerations that should be taken into account when instantiating the ICAP. First, if the output signals of the ICAP are not being used, then these signals need to be constrained with an s attribute. This attribute is a basic mapping constraint that prevents the removal of loadless or driverless signals. It places dummy loads in order to prevent the removal of the component that drives these signals, *e.g.* the ICAP. In addition, a netlist should not instantiate just the ICAP, especially if the outputs of this primitive are not used such as in the case of the proposed methodology. Otherwise, *Select IO banking rule* errors are to be encountered during the placement implementation step.

Ideally, the IO interfaces of the modules to be placed at different times in a particular PRR of a design are the same. In this case, the only bus macro consideration required is the general PR rule of placing bus macros next to all the PRR interface signals. Yet, it is often the case that the modules to be placed in a particular PRR do not have the same IO interfaces. However, in order for two or more module-based PR projects to instantiate these modules as PRMs of a PRR, the interfaces of these modules must be the same. As a result of the modification of these interfaces, PRM-shared and PRM-specific signals can be found in the top level interface of these PRMs [23]. PRM-shared signals refer to the signals

that are common to all the modules that need to be implemented in a PRR. PRM-specific signals refer to the signals that are not common to all these modules.

The following design considerations allow a PR design to successfully instantiate modules that do not have the same interface as PRMs. First of all, these modules need to be wrapped in such a way so that they share the same interface. These newly generated PRMs then need to be instantiated as follows. Since bus macros cannot be removed during implementation, a bus macro should connect both PRM-specific signals and at least one PRM-shared signal of a PRM to the static and global logic. This simple PR design consideration prevents seemingly baseless bus macro errors from occurring. If PRM-shared signals are not used by a PRM, bus macro errors will be encountered. These errors will indicate which bus macro instantiation(s) the trimming process is trying to remove. The s attribute may be used here to prevent signals connected to these instantiations from being removed. Note that this workaround will place dummy loads, and therefore will require additional logic.

While bus macro related issues are being fixed, it is important to ensure that the static design remains the same across PR projects. Bus macro warnings are likely to be encountered during the implementation of the PR projects, since some signals in the bus macro instantiations will be considered unused logic and therefore be removed. To minimize the number of bus macro related warnings during place and route, bus macros should be reused across PRM instantiations as much as possible.

Chapter 7

Results

The results of the prototype are presented in this section. First, a quantitative analysis regarding the FPGA resource utilization in the prototype is presented. Then, the performance of the prototype is discussed to determine if the processing speed requirements of the application are met.

The FPGA resource usage of the PR CSC engine prototype is provided in Table 7.1. The PRR of this design uses one third of the slices available in the array. In addition, the logic required to perform partial reconfiguration in this prototype employs less than 0.5 percent of the slices in the FPGA. Therefore, the FPGA slices required to implement the CSC engine would increase approximately by 32.5 percent if the implementation of the CSC engine did not use some sort of run-time reconfiguration. In fact, this implementation would require an FPGA denser than the one of the prototype, since 73 percent of the FPGA slices are required to just implement a single PRM and the static modules. The proposed methodology provides one way to use run-time reconfiguration without having to modify

Table 7.1: CSC engine implementation results (XC2VP30).

Feature	PRR	Static Region and PRR	Available Resources
Slices	4,407	10,035	13,696
BRAMs	60	92	136
Slice Flip Flops	1,399	4,313	27,392
4 input LUTs	7,214	16,732	27,392
Max. clock rate	50MHz		

Table 7.2: Reconfiguration performance.

User Logic	Bit-stream Size	Reconfiguration Time w/TestRig (measured)	Reconfiguration Time raw design (calculated)
PRR	465 KB	550.4 ms	9.5 ms
Full FPGA	1414 KB	n/a	29.0 ms

the external interface of the design.

On the other hand, implementing a design originally developed for an ASIC in an FPGA always has some consequences. Significant performance degradation in the maximum frequency of the CSC engine has been the most important hit encountered in the prototype implementation (from 166 MHz to 50 MHz). The inherent degradation of porting a design from an ASIC to an FPGA implementation can be minimized by modifying the modules in this design in order to leverage the FPGA architecture (*e.g.*, deeper pipelining). Nonetheless, in spite of the performance degradation when compared to the original ASIC implementation, the prototype system could still be used in some of the commercial printers, namely low-end printers, in which the CSC engine is not driven at full speed. In this context, a realistic goal for the prototype system is to reconfigure the PRR, load the CLUT values and process an 8.5 inch by 11 inch image within one second.

Table 7.2 shows both the measured and the calculated reconfiguration times of the entire FPGA design and that of the PRR. This comparison table is provided merely for completeness, since most MC systems would not tolerate the entire FPGA user logic to be unprogrammed even if it is just for 29ms. Yet, even if full dynamic reconfiguration is acceptable for a given application, partial reconfiguration would enable much faster reconfiguration times as long as the entire FPGA user logic does not need to be reconfigured dynamically. Compared to full time reconfiguration, partial reconfiguration enables the user logic on the CSC design to be reconfigured three times faster. Table 7.3 reports the measured and the calculated processing times for loading CLUT values and converting different image sizes in the prototype system. The calculated reconfiguration times have been obtained based on the maximum speed of the ICAP of the Virtex II-Pro FPGA, which is 50MHz [32].

Table 7.3: CSC engine performance.

Action	Number of CSC Cycles	Processing Time w/TestRig (measured)	Processing Time raw design (calculated)
Load CLUTs	6.8 K	7.9 ms	0.14 ms
160x120 image	19.2 K	22.2 ms	0.38 ms
8.5”x11” image	33.7 M	n/a	0.67 sec

The calculated processing times have been obtained based on Xilinx's post place and route static timing analysis of the PR CSC design, which reports a maximum speed of 50MHz also. This match in maximum frequency of operations is optimal for reusing an existing interconnect to send the PR data to the design. It should be mentioned here that an 8.5”x11” image was not processed on the TestRig only because the logic analyzer used to sample the CSC engine output data did not have enough memory.

According to the calculated values, reconfiguring the PRR, loading the CLUT, and processing a full page takes 683ms. This is well within the target of one second. Note that the reconfiguration and processing times measured are about 60 times slower than that of the calculated speed of the prototype. This is due to the relatively low speed of the hardware used to emulate the actual components that drive the CSC engine in a printer, namely the TestRig. Nevertheless, the times measured are faster than many of the results obtained by other methodologies proposed in the literature. For instance, the XAPP433 application note [31] describes a partial reconfigurable platform that is only capable of downloading the reconfiguration data to the ICAP at 500 KB/s. This rate is 100 times lesser than the ICAP one. In addition, the maximum reconfiguration speed of the partial reconfigurable platform proposed by Lagger *et al.* [20] is just over 200 KB/s.

Chapter 8

Conclusions

We have presented a methodology that enables self-partial reconfiguration in mature MC systems through additional yet negligible logic and firmware modifications inside and outside the FPGA of these systems respectively. The main contribution of the proposed methodology is that the connections amongst the *integrated circuits* (ICs) of these systems do not need to be modified in order to enable partial reconfiguration, and therefore the optimal use of FPGA resources in these systems. This allows new hardware-based algorithms to be devised in MC systems where the array resources are scarce and PCB board modifications not possible.

In addition, this thesis presents a prototype that has two main purposes. First, it demonstrates the successful application of the proposed methodology where the interface of the FPGA-based design does not change in order for partial reconfiguration features to be incorporated in the MC system. Second, this prototype has also shown that it is feasible to replace ASIC designs with FPGA-based implementations provided that there is some tolerance in terms of circuit performance and that the design has some mutually exclusive modules. The flexibility of programmable logic like FPGAs opens the possibility of rapid time-to-market development projects in these applications.

Furthermore, the results obtained from the prototype developed have demonstrated the viability of the proposed methodology. According to Xilinx's post place and route static timing analysis, partially reconfiguring the PRR of the prototype, loading the CLUTs required by any conversion after partial reconfiguration, and processing a small image can

be achieved in half a second in the prototype system. This speed is well within the target of one second set by the speed requirements of commercial low-end HP printers, which were originally thought of as the products that could use the FPGA-based CSC engine prototyped in this research.

Nonetheless, when compared to the CSC engine prototype maximum frequency of operation, the speed at which this prototype has been tested in hardware is still slow. Therefore, future work will not only explore FPGA-oriented optimizations but also aim to further increase the speed of the TestRig board. The motivations behind additional FPGA-oriented optimizations are to further reduce the resources required to implement the CSC engine on a FPGA and to narrow the gap in throughput between the FPGA-based implementation and the original ASIC. The optimizations in the roadmap to accomplish these goals are deeper pipelining and *partial evaluation* [9, 27] respectively. In addition, a DMA-based rather than the current processor-based approach can be used to increase the speed at which the TestRig board sends data to the CSC board. Note that this enhancement would not require any changes to the bill of materials of the current prototype hardware.

References

[1] Altera. Standard cell ASIC to FPGA design methodology and guidelines. Application Note [Online]. Available: http://www.altera.com/literature/an/an311.pdf, June 2008.

[2] Brandon Blodget, Philip James-Roxby, Eric Keller, Scott McMillan, and Prasanna Sundararajan. A self-reconfiguring platform. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Proc. Int. Conf. Field Programmable Logic and Applications (FPL)*, volume 2778 of *Lecture Notes in Computer Science*, pages 565–574. Springer, 2003.

[3] Brandon Blodget, Scott McMillan, and Patrick Lysaght. A lightweight approach for embedded reconfiguration of FPGAs. In *Proc. Design, Automation and Test in Europe (DATE)*, page 10399. IEEE Computer Society Press, 2003.

[4] Pierre Bomel, Guy Gogniat, and Jean-Philippe Diguet. A networked, lightweight and partially reconfigurable platform. In *Proc. Int. Symp. Applied Reconfigurable Computing (ARC)*, pages 318–323, 2008.

[5] Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. A dynamic reconfiguration run-time system. In Kenneth L. Pocek and Jeffrey Arnold, editors, *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*, pages 66–75, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[6] Javier Castillo, Pablo Huerta, Victor Lopez, and Jose Ignacio Martinez. A secure self-reconfiguring architecture based on open-source hardware. In *Proc. Int. Conf. Reconfigurable Computing and FPGAs (ReConFig)*, page 10. IEEE Computer Society Press, 2005.

[7] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[8] D. Curd. Dynamic reconfiguration of RocketIO MGT attributes. Application Note [Online]. Available: http://www.xilinx.com/support/documentation/-applicationnotes/xapp660.pdf, February 2004.

[9] Andre DéHon, Joshua Adams, Michael DeLorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton. Design patterns for reconfigurable computing. In *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*, pages 13–23. IEEE Computer Society Press, 2004.

[10] J.P. Delahaye, G. Gogniat, C. Roland, and P. Bomel. Software radio and dynamic reconfiguration on a DSP/FPGA platform. *Frequenz*, 58(8):152–159, 2004.

[11] N. Dorairaj. PlanAhead software as a platform for partial reconfiguration. Xcell Journal [Online]. Available: http://www.xilinx.com/publications/xcellonline/xcell55/-xcpdf/xcprmethod55.pdf, December 2005.

[12] Ryan J. Fong, Scott J. Harper, and Peter M. Athanas. A versatile framework for FPGA field updates: An application of partial self-reconfiguration. In *Proc. IEEE Int. Workshop Rapid System Prototyping (RSP)*, page 117. IEEE Computer Society Press, 2003.

[13] Phil Green and Lindsay MacDonald, editors. *Colour Engineering: Achieving Device Independent Colour*. John Wiley and Sons Ltd., 2002.

[14] M. Hübner, L. Braun, D. Gohringer, and J. Becker. Run-time reconfigurable adaptive multilayer network-on-chip for FPGA-based systems. In *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, pages 1–6, April 2008.

[15] IEC/IEEE. *Behavioural languages - Part 1-1: VHDL language reference manual*, first edition, 2004. IEC 61691-1-1, IEEE 1076.

[16] IEC/IEEE. *Behavioural languages - Part 4: Verilog hardware description language*, first edition, October 2004. IEC 61691-4, IEEE 1364.

[17] James M. Kasson, Sigfredo I. Nin, Wil Plouffe, and James L. Hafner. Performing color space conversions with three-dimensional linear interpolation. *Journal of Electronic Imaging*, 4(3):226–250, 1995.

[18] Ryusuke Konishi, Hideyuki Ito, Hiroshi Nakada, Akira Nagoya, Kiyoshi Oguri, Norbert Imlig, Tsunemichi Shiozawa, Minoru Inamori, and Kouichi Nagami. PCA-1: A

fully asynchronous self-reconfigurable LSI. In *Proc. Int. Symp. Advanced Research in Asynchronous Circuits and Systems*, pages 54–61. IEEE Computer Society Press, March 2001.

[19] Yana E. Krasteva, Ana B. Jimeno, Eduardo de la Torre, and Teresa Riesgo. Straight method for reallocation of complex cores by dynamic reconfiguration in Virtex II FPGAs. In *Proc. IEEE Int. Workshop Rapid System Prototyping (RSP)*, pages 77–83. IEEE Computer Society Press, 2005.

[20] A. Logger, A. Upegui, E. Sanchez, and I. Gonzalez. Self-reconfigurable pervasive platform for cryptographic application. In *Proc. Int. Conf. Field Programmable Logic and Applications (FPL)*, pages 1–4, Aug. 2006.

[21] Théodore Marescaux, Andrei Bartic, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Interconnection networks enable fine-grain dynamic multi-tasking on FPGAs. In *Proc. Int. Conf. Field Programmable Logic and Applications (FPL)*, pages 795–805, London, UK, 2002. Springer-Verlag.

[22] Juanjo Noguera and Robert Esser. Application-driven research in partial reconfiguration. [Online]. Available: http://ce.et.tudelft.nl/cecoll/slides/07/0524noguera.pdf, March 2007.

[23] Sreenivas Patil. Reconfigurable hardware for color space conversion. Master's thesis, Dept. Electrical Engineering., Rochester Institute of Technology, May 2008.

[24] Thilo Pionteck, Roman Koch, and Carsten Albrecht. Applying partial reconfiguration to networks-on-chips. In *Proc. Int. Conf. Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, 2006.

[25] N. Pete Sedcole, Brandon Blodget, Tobias Becker, James Anderson, and Patrick Lysaght. Modular partial reconfiguration in Virtex FPGAs. In Tero Rissa, Steven J. E. Wilton, and Philip Heng Wai Leong, editors, *Proc. Int. Conf. Field Programmable Logic and Applications (FPL)*, pages 211–216. IEEE, 2005.

[26] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in Virtex FPGAs. *IEE Proceedings - Computers and Digital Techniques*, 153(3):157–164, 2006.

[27] S. Singh, J. Hogg, and D. McAuley. Expressing dynamic reconfiguration by partial evaluation. In J. Arnold and K. L. Pocek, editors, *Proc. IEEE Symp. Field-Programmable Custom Computing Machines (FCCM)*, pages 188–194, Napa, CA, April 1996.

[28] Christoph Steiger, Herbert Walder, and Marco Platzner. Heuristics for online scheduling real-time tasks to partially reconfigurable devices. In Peter Y. K. Cheung, George A. Constantinides, and José T. de Sousa, editors, *Proc. Int. Conf. Field Programmable Logic and Applications (FPL)*, volume 2778 of *Lecture Notes in Computer Science*, pages 575–584. Springer, 2003.

[29] Jens Thorvinger. Dynamic partial reconfiguration of an FPGA for computational hardware support. Master's thesis, Dept. Electroscience, Lund Institute of Technology, June 2004.

[30] Grant B. Wigley, David A. Kearney, and David Warren. Introducing ReConfigME: An operating system for reconfigurable computing. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Proc. Int. Conf. Field Programmable Logic and Applications (FPL)*, volume 2438 of *Lecture Notes in Computer Science*, pages 687–697. Springer, 2002.

[31] Xilinx. Web server design using MicroBlaze soft processor. Application Note [Online]. Available: http://www.xilinx.com/, October 2006.

[32] Xilinx. Virtex II-Pro FPGA user guide. Application Note [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug012.pdf, November 2007.

Appendix A

TestRig Board Implementation Details

The firmware that runs on the TestRig board is provided in Figures A.1 through A.7. The hardware specification of the TestRig board consists of the following:

Target Board: Xilinx XUP Virtex-II Pro Development System Rev C

Family: virtex2p

Device: xc2vp30

Package: ff896

Speed Grade: -7

Processor: PPC 405

Processor clock frequency: 300 MHz

Bus clock frequency: 100 MHz

Debug interface: FPGA JTAG

On Chip Memory : 76 KB

Total Off Chip Memory : 256 MB

- DDR_SDRAM_32Mx64 Single Rank = 256 MB

```
/*
 * The main purpose of this Xilinx EDK application is to read test files from a CF card,
 * parse them into CSC packets, and store them in DDR memory so that a complete CSC frame
 * can be then sent from this DDR memory to an IO device of the TestRig board. The name of
 * this IO device is dut_data, which is memory-mapped making it very easy for a processor
 * to send data to it.
 */

// Common libraries
#include "xparameters.h"
#include "stdio.h"
#include "xsysace.h"
#include "sysace_header.h"
#include "xbasic_types.h"
#include "xgpio.h"
#include "gpio_header.h"
#include "xuartlite.h"

// Memory-mapped definitions for dut_data and pattern_gen (test only) IO devices
#include "pattern_gen.h"
#include "dut_data.h"

// Xilinx FAT library
#include "sysace_stdio.h"

//====================================================
#define PRM_FILE_SIZE_MAX    500000
#define NUM_CSC_SAMPLES_MAX  600000

// PR data buffer in DDR - for testing only
Xuint32 icapdata[PRM_FILE_SIZE_MAX] __attribute__ ((section ("TESTDATA"))) = { 0 };

// CSC packet structure
typedef struct {
    Xuint32 cntrl_plus_addr;
    Xuint32 reg_data;
    Xuint32 pix_data_0_1;
    Xuint32 pix_data_2_3;
} CSCDATASTRUCT;

// CSC frame buffers in DDR
CSCDATASTRUCT cscimg1[NUM_CSC_SAMPLES_MAX] __attribute__ ((section ("TESTDATA"))) = { 0 };
CSCDATASTRUCT cscimg2[NUM_CSC_SAMPLES_MAX] __attribute__ ((section ("TESTDATA"))) = { 0 };

// UART instance
XUartLite uart1;

// Global variables
int image_loaded;
int partial_bitstream_loaded;

// Minimalistic get string implementation
void read_usr_input(char* user_input) {
   int  user_input_cnt = 0;
   int  endofline      = 0;
   int  char_received;
   char temp_char[2];

   //initializing variables
   temp_char[1] = '\0';

   while (!endofline) {
      // initializing variables
      char_received = 0;
```

Figure A.1: TestRig board firmware - part 1.

```
      // poll the uart until a character is received
      while (char_received == 0) {
         char_received = XUartLite_Recv (&uart1, &temp_char[0], 1);
      }

      // is the byte received a carriage return
      if (temp_char[0] == 13) {
         user_input[user_input_cnt] = '\0';
         endofline = 1;
         // echo received ascii
         xil_printf("%s",temp_char);
      }
      // is the byte received a backspace
      else if (temp_char[0] == 8) {
         if (user_input_cnt != 0) {
            user_input_cnt--;
            // echo received ascii
            xil_printf("%s",temp_char);
         }
      }
      // store character received
      else {
         user_input[user_input_cnt] = temp_char[0];
         user_input_cnt++;
         // echo received ascii
         xil_printf("%s",temp_char);
      }
   }
}

// Function that sends a stored CSC frame to the dut_data IO device
void process_img(int csc_sample_size,CSCDATASTRUCT* cscimg) {
   int cscdatacount;
   xil_printf("\n\n\rExecuting process_img()...\n\r");

   //send dut test data
   cscdatacount = 0;
   while (cscdatacount < csc_sample_size) {
      *(volatile Xuint32 *)(XPAR_DUT_DATA_1_BASEADDR) = cscimg[cscdatacount].cntrl_plus_addr;
      *(volatile Xuint32 *)(XPAR_DUT_DATA_1_BASEADDR) = cscimg[cscdatacount].reg_data;
      *(volatile Xuint32 *)(XPAR_DUT_DATA_1_BASEADDR) = cscimg[cscdatacount].pix_data_0_1;
      *(volatile Xuint32 *)(XPAR_DUT_DATA_1_BASEADDR) = cscimg[cscdatacount++].pix_data_2_3;
   }
}

// Function that sends PR data to the pattern_gen IO device - test function
void reconfig_engine(int reconfig_data_size) {
   int var;
   xil_printf("\n\n\rExecuting reconfig_engine()...\n\r");

   //signal ICAP beginning of payload
   *(volatile Xuint32 *)(XPAR_PATTERN_GEN_1_BASEADDR+4) = 0x1;

   //send payload
   var = 0;
   while (var < reconfig_data_size) {
      *(volatile Xuint32 *)(XPAR_PATTERN_GEN_1_BASEADDR) = icapdata[var++];
   }

   //signal ICAP end of payload
   *(volatile Xuint32 *)(XPAR_PATTERN_GEN_1_BASEADDR+4) = 0x0;
}
```

Figure A.2: TestRig board firmware - part 2.

```
// Function that read a particular user-supplied PR data file from a CF card into DDR memory
// This is test function
void load_reconfig_data(char* reconfig_data_file_name, int reconfig_data_size) {
   SYSACE_FILE * prmfile;              // pointer to prm file
   char prmbuffer[2];        //Buffer to hold the file contents
   int tcount;

   // display a message showing current step of the file read process
   xil_printf("\n\n\rExecuting load_reconfig_data()...\n\r");

   // try to open prm file
   prmfile = sysace_fopen(&reconfig_data_file_name[0], "r");

   // if there was a problem opening the file exit program
   // if there was no problem, continue processing
   if (prmfile == 0) {
      print("\n\rfile open failed.\n\r");
      return;
   }
   else {
      // the file was opened sucessfully, try to read from it
      // appened a NULL byte to the end buffer space
      prmbuffer[2] = '\0';

      //read from file
      for (tcount = 0; tcount < reconfig_data_size ; tcount++) {
         sysace_fread(prmbuffer, 1, 2, prmfile);
         icapdata[tcount] = strtoul(prmbuffer,NULL,16);
      }

      //closing PRM file
      sysace_fclose(prmfile);

      xil_printf("\r\nicapdata %d: 0x%x", tcount-1, icapdata[tcount-1]);

      print("\r\n");
      //partial bitstream loaded successfully
      partial_bitstream_loaded = 1;
   }
}

// Function that read a particular user-supplied test file from a CF card into DDR memory
void load_img (char* img_file_name, int csc_sample_size, CSCDATASTRUCT* cscimg) {
   SYSACE_FILE * cscdatafile; // pointer to csc file
   int tcount;
   int cscdatacount;

   // display a message showing current step of the file read process
   xil_printf("\n\n\rExecuting load_img()...\n\r");

   // try to open cscimg file
   cscdatafile = sysace_fopen(&img_file_name[0], "r");

   // if there was a problem opening the file exit program
   // if there was no problem, continue processing
   if (cscdatafile == 0) {
      print("\n\rfile open failed.\n\r");
      return;
   }
   else {
      char csccntrlbuf1[9];
      char csccntrlbuf2[2];
```

Figure A.3: TestRig board firmware - part 3.

```
char csccntrlbuf3[5];
char csccntrlbuf4[9];
char ignorespace;

// the file was opened sucessfully, try to read from it
// append a NULL byte at the end of each temp buffer
csccntrlbuf1[8] = '\0';
csccntrlbuf2[1] = '\0';
csccntrlbuf3[4] = '\0';
csccntrlbuf4[8] = '\0';

for (cscdatacount = 0 ; cscdatacount < csc_sample_size; cscdatacount++) {
   //read from file (in_pipeline_enable, in_addr_valid, in_reg_write)
   for (tcount = 0; tcount < 3 ; tcount++) {
      // read signal value
      sysace_fread(&csccntrlbuf1[tcount], 1, 1, cscdatafile);

      // disregard space
      sysace_fread(&ignorespace, 1, 1, cscdatafile);
   }

   //read from file (in_reg_addr [17:16])
   sysace_fread(&csccntrlbuf2[0], 1, 1, cscdatafile);

   //store csc data (cntrl_plus_addr[17:16])
   cscimg[cscdatacount].cntrl_plus_addr = strtoul(csccntrlbuf2,NULL,16) << 16;

   //read from file (in_reg_addr [15:0])
   sysace_fread(&csccntrlbuf3[0], 1, 4, cscdatafile);

   //store csc data (cntrl_plus_addr[15:0])
   cscimg[cscdatacount].cntrl_plus_addr = cscimg[cscdatacount].cntrl_plus_addr |
   strtoul(csccntrlbuf3,NULL,16);

   // disregard space
   sysace_fread(&ignorespace, 1, 1, cscdatafile);

   //read from file (in_reg_wdata [31:0])
   sysace_fread(&csccntrlbuf4[0], 1, 8, cscdatafile);

   //store csc data (reg_data [31:0])
   cscimg[cscdatacount].reg_data =  strtoul(csccntrlbuf4,NULL,16);

   // disregard space
   sysace_fread(&ignorespace, 1, 1, cscdatafile);

   //read from file (In_PreCscAck, In_PreCscEol, In_PreCscEop, In_PreCscOt,
   //In_PreCscNop)
   for (tcount = 3; tcount < 8 ; tcount++) {
      // read signal value
      sysace_fread(&csccntrlbuf1[tcount], 1, 1, cscdatafile);

      // disregard space
      sysace_fread(&ignorespace, 1, 1, cscdatafile);
   }

   //store csc data (cntrl_plus_addr[31:24])
   cscimg[cscdatacount].cntrl_plus_addr = cscimg[cscdatacount].cntrl_plus_addr |
   (strtol(csccntrlbuf1,NULL,2) << 24);

   //read from file (In_PreCscData0[15:0])
   sysace_fread(&csccntrlbuf3[0], 1, 4, cscdatafile);

   //store csc data (pix_data_0_1 [31:16])
   cscimg[cscdatacount].pix_data_0_1 =  strtoul(csccntrlbuf3,NULL,16) << 16;
```

Figure A.4: TestRig board firmware - part 4.

```
      //disregard space
      sysace_fread(&ignorespace, 1, 1, cscdatafile);

      //read from file (In_PreCscData3[15:0])
      sysace_fread(&csccntrlbuf3[0], 1, 4, cscdatafile);

      //store csc data (pix_data_2_3 [15:0])
      cscimg[cscdatacount].pix_data_2_3 = cscimg[cscdatacount].pix_data_2_3 |
      strtoul(csccntrlbuf3,NULL,16);

      // disregard space
      sysace_fread(&ignorespace, 1, 1, cscdatafile);

      // read signal value
      sysace_fread(&csccntrlbuf2[0], 1, 1, cscdatafile);

      //store csc data (cntrl_plus_addr[18])
      cscimg[cscdatacount].cntrl_plus_addr = cscimg[cscdatacount].cntrl_plus_addr |
      (strtol(csccntrlbuf2,NULL,2) << 18);

      // disregard space
      sysace_fread(&ignorespace, 1, 1, cscdatafile);

      // disregard space
      sysace_fread(&ignorespace, 1, 1, cscdatafile);
   }

   //closing PRM file
   sysace_fclose(cscdatafile);

   //print last csc packet
   xil_printf("\r\nCSC cntrl_plus_addr: 0x%x\r\n",cscimg[csc_sample_size-1].cntrl_plus_addr);
   xil_printf("\r\nCSC reg_data       : 0x%x\r\n",cscimg[csc_sample_size-1].reg_data);
   xil_printf("\r\nCSC pix_data_0_1   : 0x%x\r\n",cscimg[csc_sample_size-1].pix_data_0_1);
   xil_printf("\r\nCSC pix_data_2_3   : 0x%x\r\n",cscimg[csc_sample_size-1].pix_data_2_3);

   //image loaded successfully
   image_loaded = 1;
   return;
}

// Program execution begin here
int main (void) {
   char    user_option_str [2];
   int     num_char_received;
   int     user_option;
   int     csc_sample_size_1;
   int     csc_sample_size_2;
   int     partial_bitstream_size;
   Xuint8  menu_options [] = "Firmware options:\r\n \
   1. Load Image\r\n \
   2. Process Image\r\n \
   3. Load Reconfig Data\r\n \
   4. Reconfig Engine";

   // Some global initializations
   XUartLite_Initialize  (&uart1, XPAR_RS232_UART_1_DEVICE_ID);
   image_loaded = 0;
   partial_bitstream_loaded = 0;

   while (1) {
      // initialize variables
      num_char_received = 0;
      // display menu options
      xil_printf("\r\n%s",&menu_options[0]);
```

Figure A.5: TestRig board firmware - part 5.

```
// prompt user to enter an option
xil_printf("\r\nEnter option: ");

// wait for user's input
read_usr_input(&user_option_str[0]);

// covert user option in ascii to integer
user_option = strtol(user_option_str,NULL,16);

// process user's request
switch (user_option)
{
   char img_file_name[13];
   char csc_sample_size_str[6];
   char partial_bitstream_name[13];
   char partial_bitstream_size_str[7];
   char img_number[2];

   case 1:
      //request img file name
      xil_printf("\r\nEnter image file name: ");

      //wait for user input
      read_usr_input(&img_file_name[0]);

      //request csc sample size
      xil_printf("\r\nEnter csc sample size: ");

      //wait for user input
      read_usr_input(&csc_sample_size_str[0]);

      //select image number
      xil_printf("\r\nEnter image number (1|2): ");

      //wait for user input
      read_usr_input(&img_number[0]);

      //load image
      if (img_number[0] == '1'){
         //convert user input to integer
         csc_sample_size_1 = strtol(csc_sample_size_str,NULL,10);
         load_img(&img_file_name[0], csc_sample_size_1, cscimg1);
      }
      else {
         //convert user input to integer
         csc_sample_size_2 = strtol(csc_sample_size_str,NULL,10);
         load_img(&img_file_name[0], csc_sample_size_2, cscimg2);
      }
      break;

   case 2:
      if (image_loaded == 1) {
         //select image number
          xil_printf("\r\nEnter image number (1|2): ");

         //wait for user input
         read_usr_input(&img_number[0]);
         if (img_number[0] == '1'){
            process_img(csc_sample_size_1, cscimg1);
         }
         else {
            process_img(csc_sample_size_2, cscimg2);
         }
      }
```

Figure A.6: TestRig board firmware - part 6.

```
            else {
               xil_printf("\r\nImage not loaded: ");
            }
            break;

        case 3:
            //request partial bitstream name
            xil_printf("\r\nEnter partial bitstream file name: ");

            //wait for user input
            read_usr_input(&partial_bitstream_name[0]);

            //request partial bitstream size
            xil_printf("\r\nEnter partial bitstream file size: ");

            //wait for user input
            read_usr_input(&partial_bitstream_size_str[0]);

            //convert user input to integer
            partial_bitstream_size = strtol(partial_bitstream_size_str,NULL,10);

            //load partial reconfiguration data
            load_reconfig_data(&partial_bitstream_name[0], partial_bitstream_size);
            break;

        case 4:
           if (partial_bitstream_loaded == 1) {
              reconfig_engine(partial_bitstream_size);
           }
           else {
              xil_printf("\r\nPartial bitstream not loaded: ");
           }
           break;

        default:
           xil_printf("\r\nInvalid option");
        }
    }
    return 0;
}
```

Figure A.7: TestRig board firmware - part 7.

Appendix B

CSC Board Implementation Details

The CSC engine prototype of the CSC board has been synthesized using Xilinx ISE 8.2.03. The RTL produced by this synthesis has been floorplanned using Xilinx PlanAhead 8.2.10 and implemented using Xilinx ISE 8.2.01_PR_07b.

The hardware specification of the CSC board consists of the following:

Target Board: Xilinx XUP Virtex-II Pro Development System Rev C

Family: virtex2p

Device: xc2vp30

Package: ff896

Speed Grade: -7

System clock frequency: 100 MHz

Debug interface: FPGA JTAG

In addition to the constraints required to route the top-level IO signals of the prototype to the expansion headers on the CSC board, the Clk signal generated by the depacketizer module has been assigned a global routing resource such that the skew of this gated clock, which is used as the reference clock in the unmodified portions of the CSC design, is minimized. To assign a global routing resource to this signal, the following instantiation in the top-level module and constraint in the *user constraint file* (UCF) are required.

In Verilog top-level module:

// Buffering clock

BUFG clock_bufg_inst (

.I(Clk),

.O(Clk_buf)

);

In UCF file:

BFUG constraints

INST ”clock_bufg_inst” LOC = BUFGMUX1S;

Furthermore, the place and route process in Xilinx ISE 8.2.01_PR_07b has been set to medium, since a higher effort level would have taken a very long time to implement due to the complexity of both the CSC engine design and the PR implementation flow.

Finally, although the source files of the original CSC engine modules that have been modified to support PR cannot be shown due to proprietary issues, the source file of the new module added in the prototype, *i.e.* the ICI module, is provided in Figures A.1 through A.7

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The purpose of this module is to send the siphoned PR data to the ICAP. While partial reconfiguration
// is taking place, this module stalls the CSC engine pipeline.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module icap_eai(
   Clk,
   nReset,
   PipelineGo,
   PREnable,
   PRData,
   PRDataCount,
   PRInProg,
   PRCS,
   PRDone);

input         Clk;
input         nReset;
input         PipelineGo;
input         PREnable;
input  [ 7:0] PRData;
input  [18:0] PRDataCount;
output        PRInProg;
output        PRCS;
output        PRDone;

// Combinational output signal
wire        PRCS                ;

// Registered output signals
reg         PRDone              ;
reg         PRInProg            ;

// Internal ICAP control signals
wire        Icap_Busy           ;
wire [ 7:0] Icap_Output         ;
reg         Icap_Ce             ;
reg         Icap_Write          ;
reg         Icap_Busy_Dummy_Load  ;
reg  [ 7:0] Icap_Output_Dummy_Load ;

// synthesis attribute s of Icap_Busy_Dummy_Load is "yes";
// synthesis attribute s of Icap_Output_Dummy_Load is "yes";

// Miscellaneous internal combinational and registered signals and buses
wire        PRByteCntReached    ;
reg  [18:0] PRByteCounter       ;
reg         PRByteCounterEnable  ;
reg         PRCSDelay1          ;
reg  [ 7:0] PRDataDelay1        ;
reg  [ 7:0] PRDataDelay2        ;

// State machines
reg  [ 1:0] reconfigsm;
// ICI control parameters
parameter RECONFIG_IDLE      = 2'b00;
parameter RECONFIG_CYPHENING = 2'b01;
parameter RECONFIG_DONE      = 2'b10;
parameter RECONFIG_WAIT      = 2'b11;

reg  [ 1:0] icap_cntrl_sm;
// ICAP control parameters
parameter ICAP_IDLE      = 2'b00;
parameter ICAP_START     = 2'b01;
parameter ICAP_RECONFIG  = 2'b10;
parameter ICAP_STOP      = 2'b11;
```

Figure B.1: ICI module - part 1.

```
// ICI control logic (sequential)
always @(posedge Clk or negedge nReset) begin
  if (!nReset) begin
    reconfigsm <= RECONFIG_IDLE;
  end else begin
    if (PipelineGo) begin
                case (reconfigsm)
                  RECONFIG_IDLE:
                    if (PREnable) begin
                      reconfigsm <= RECONFIG_CYPHENING;
                    end else begin
                      reconfigsm <= RECONFIG_IDLE;
                    end

                  RECONFIG_CYPHENING:
                    if (PRByteCntReached) begin
                      reconfigsm <= RECONFIG_DONE;
                    end else begin
                      reconfigsm <= RECONFIG_CYPHENING;
                    end

                  RECONFIG_DONE:
                    reconfigsm <= RECONFIG_WAIT;

                  RECONFIG_WAIT:
                    reconfigsm <= RECONFIG_IDLE;

                  default:
                    reconfigsm <= RECONFIG_IDLE;
                endcase
    end
  end
end

// ICI control logic (combinatorial)
always @(reconfigsm or PREnable) begin
  case (reconfigsm)
    RECONFIG_IDLE:
    begin
      if (PREnable) begin
        PRInProg    <= 1'b1;
      end else begin
        PRInProg    <= 1'b0;
      end
      PRByteCounterEnable <= 1'b0;
      PRDone          <= 1'b0;
    end

    RECONFIG_CYPHENING:
    begin
      PRInProg    <= 1'b1;
      PRByteCounterEnable <= 1'b1;
      PRDone          <= 1'b0;
    end

    RECONFIG_DONE:
    begin
      PRInProg    <= 1'b0;
      PRByteCounterEnable <= 1'b0;
      PRDone          <= 1'b1;
    end
```

Figure B.2: ICI module - part 2.

```
      RECONFIG_WAIT:
      begin
        PRInProg    <= 1'b0;
        PRByteCounterEnable <= 1'b0;
        PRDone          <= 1'b0;
      end

      default:
      begin
        PRInProg    <= 1'b0;
        PRByteCounterEnable <= 1'b0;
        PRDone          <= 1'b0;
      end
    endcase
end

// Universal 19-bit up counter
always @(posedge Clk or negedge nReset) begin
  if (!nReset) begin
    PRByteCounter <= 19'h00000;
  end else if (PRDone) begin
    PRByteCounter <= 19'h00000;
  end else if (PipelineGo && PRByteCounterEnable) begin
    PRByteCounter <= PRByteCounter + 1;
  end
end

// Universal 19-bit comparator
assign PRByteCntReached = PRByteCounter == (PRDataCount - 2);

// PRCS delay
always @(posedge Clk or negedge nReset)
begin
  if (!nReset) begin
    PRCSDelay1          <=  1'b0;
  end else begin
    PRCSDelay1          <=  PRCS;
  end
end

// PRData delay
always @(posedge Clk or negedge nReset)
begin
  if (!nReset) begin
    PRDataDelay1        <=  8'h00;
    PRDataDelay2        <=  8'h00;
  end else begin
    PRDataDelay1        <=  PRData;
    PRDataDelay2        <=  PRDataDelay1;
  end
end

// ICAP control logic (sequential)
always @(posedge Clk or negedge nReset)
begin
  if (!nReset) begin
    icap_cntrl_sm       <=  ICAP_IDLE;
  end else begin
    case (icap_cntrl_sm)
      ICAP_IDLE:
      begin
        if (PRCS) begin
          icap_cntrl_sm <=  ICAP_START;
        end
      end
```

Figure B.3: ICI module - part 3.

```
        ICAP_START:
          icap_cntrl_sm    <=  ICAP_RECONFIG;

        ICAP_RECONFIG:
        begin
          if (!PRCSDelay1) begin
            icap_cntrl_sm <=  ICAP_STOP;
          end
        end

        ICAP_STOP:
          icap_cntrl_sm    <=  ICAP_IDLE;

        default:
          icap_cntrl_sm    <=  ICAP_IDLE;
      endcase
    end
  end

  // ICAP control logic (combinational)
  always @(icap_cntrl_sm)
  begin
    case (icap_cntrl_sm)
      ICAP_IDLE:
      begin
        Icap_Write         <=  1'b1;
        Icap_Ce            <=  1'b1;
      end

      ICAP_START:
      begin
        Icap_Write         <=  1'b0;
        Icap_Ce            <=  1'b1;
      end

      ICAP_RECONFIG:
      begin
        Icap_Write         <=  1'b0;
        Icap_Ce            <=  1'b0;
      end

      ICAP_STOP:
      begin
        Icap_Write         <=  1'b0;
        Icap_Ce            <=  1'b1;
      end
    endcase
  end

  // ICAP Dummy Loads (to prevent removal of ICAP primitive)
  always @(posedge Clk or negedge nReset)
  begin
    if (!nReset) begin
      Icap_Busy_Dummy_Load   <=  1'b0;
      Icap_Output_Dummy_Load <=  8'h00;
    end
    else begin
      Icap_Busy_Dummy_Load   <=  Icap_Busy;
      Icap_Output_Dummy_Load <=  Icap_Output;
    end
  end

  // Miscellaneous combinational output
  assign PRCS = PipelineGo && PREnable && !PRDone;
```

Figure B.4: ICI module - part 4.

```
// ICAP primitive instantiation
ICAP_VIRTEX2 ICAP_VIRTEX2_INST(
  .BUSY  (Icap_Busy),
  .O     (Icap_Output),
  .CE    (Icap_Ce),
  .CLK   (Clk),
  .I     (PRDataDelay2),
  .WRITE (Icap_Write));

endmodule
```

Figure B.5: ICI module - part 5.